Chatter: A Conversational Telephone Agent

by

Eric ThichVi Ly

B.S., Symbolic Systems
Stanford University
1991

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of

Master of Science

at the

Massachusetts Institute of Technology
June 1993

Signature of Author

Program in Media Arts and Sciences
May 13, 1993

Certified by

Christopher M. Schmandt
Principal Research Scientist, MIT Media Laboratory
Thesis Supervisor

Accepted by

Stephen A. Benton
Chairperson
Departmental Committee on Graduate Students

Chatter: A Conversational Telephone Agent

by

Eric ThichVi Ly

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on May 13, 1993
in partial fulfillment of the requirements for the degree of

Master of Science

at the

Massachusetts Institute of Technology

Abstract. Chatter is a conversational speech agent for supporting the exchange of information among members of a work group. It explores interfaces centered around the conversational and interface agent paradigms. While the recent integration of speech recognition technology into user interfaces has brought great promise for ease of use, the dreams go largely unfulfilled because speech is still used mainly in the command-driven style. In natural conversation, participants exchange thoughts fluidly and dynamically. As applications do more, users cannot remember all capabilities of the interface, so applications must take more active roles in the conversational experience by recognizing user goals and suggesting the appropriate capabilities of the program. Chatter is distinguished by its maintenance of a user-computer discourse model, which is used as a basis for inferring user interests, suggesting courses of action and alerting the user to potentially interesting information and events.

Thesis Supervisor: Christopher M. Schmandt
Title: Principal Research Scientist, MIT Media Laboratory

This work was sponsored by Sun Microsystems.

Thesis Committee

Thesis Supervisor

Christopher M. Schmandt
Principal Research Scientist
MIT Media Laboratory

Reader

Candace L. Sidner
Member of Research Staff
DEC Cambridge Research Laboratory

Reader

Pattie Maes
Assistant Professor of Media Arts and Sciences
MIT Media Laboratory

Acknowledgments

My deepest appreciation and gratitude to the people on this page, without whose support and encouragement, this project would have been much more trying:

Chris Schmandt, my adviser, for being a tough but fair parent in my post-undergraduate life. I wish to thank him for sharing with me his fine sight and enthusiasm about speech interfaces, and for having patience and understanding about the challenges of this project. Chris instilled in me the dogged perseverance needed to see tough projects through.

Candy Sidner, my reader, for her many excellent suggestions, feedback and advice on the thesis about speech and discourse. I hope that my work has been worthy of her time.

Pattie Maes, my reader, for teaching a wonderful course on agents that inspired me, and for answering my questions about machine learning and suggesting avenues of exploration.

John Etchemendy, for his enduring encouragement and support and kind friendship, all of which truly inspired me to stay the course during my adjustment to MIT. He is a great person and someone whose many excellent qualities I aspire to learn some day.

Lisa Stifelman, for her magical stress-busting pep talks and brainstorming sessions about discourse. Her thoroughness and grasp of material made me think of things I otherwise wouldn't have. Her excellent thesis was the inspiration and model for this one.

Charles Hemphill at Texas Instruments for long hours and valuable expertise on recognition issues. Charles worked hard to reconstitute his group's work into a form we could use and answered my endless questions. Without his help, this thesis could not have been done.

Don Jackson, my boss at Sun and wizard-at-large, for his positive energy and support. Don is an all-around great guy who sets the standard for future bosses.

Atty Mullins, for his brainstorming sessions on language and skill at maintaining sanity. Atty contributed his good ideas to the thesis and had the patience to listen to my odd ideas. Most of all, his calm brought a sanity that made the group an even better place to work.

Barry Arons, my officemate, for challenging me to think about speech issues and find my own interests. Barry is a very resourceful guru on speech.

A great debt to close friends over the last two years, in no particular order: Joe C., Christopher F., Monica F., Diane L., Monika G., Joe B., Tom M., Elaine F., Robert N., Michael L., Frank C., Dan R., Deb C., Chris H., Scott F., Debby H., Mike C. and Wendy Y. My housemates Takeo K., Erik M., Ann P. and Brian. I wish I could summarize the many acts of friendship they've shared. They gave me spiritual support and the perspective to sustain the importance of people, friends and living beyond school in life.

Finally, a special thanks to my parents, whose endless devotion and caring for their children inspired them to live life with a high standard.

Table of Contents

List of Figures

1. Introduction

As computers become smaller in size, it becomes more difficult for them to sport visual displays or keyboards. The use of speech as an interface modality, for input and output, has gained interest recently because it does not take up physical space to effect. What gains are made in space are lost in time; old paradigms of human-computer interaction need to be rethought. As a channel for human-computer interaction, the audio domain is relatively impoverished compared to visual mediums. Whereas a great amount of information can be presented at once on a visual display for a user to simply glance at, a similar audio display requires time to hear.

Most interactive speech-only applications today have command-style interfaces, which are usually slow and inefficient. Simply listening to a menu of commands takes time, and in order to choose anything, the user has to remember all of the available options, or know what he wants to do ahead of time and then catch the command as it is announced. If a menu is repeated, that takes time, too. Repeating a menu entails navigation, which is not instantaneous as in the visual domain because the user has to recognize what he is hearing to know where he is. The effort in listening to synthetic speech not only is time-consuming but also demands a greater cognitive load [Pis85]. In the speech domain, the command-style becomes even more problematic when the number of commands becomes large. Since the list is large, it cannot all be remembered by the user. To be presented, it needs to be spoken, taking up valuable time. One solution is to group commands into logical sets and place them into hierarchical menus. However, the process of choosing actions becomes rigid and modal[1]. Working through a set of hierarchical menus does not correspond to the way people use speech everyday.

What can be done in speech-only interfaces when the application becomes complex and many capabilities are available? How can these choices be made known to the user without costing too much time? If the ever growing group of non-trained users has to make effective use of the power and diversity the computer provides, current speech interfaces will prove to be insufficient.

1. *Modal* is used to describe an interaction in which the user is forced to operate in a limited context imposed by the system before any other operations can be done. One common example of a mode is a dialog box which asks the user a question. He has to answer *OK* or *cancel* before proceeding. The user is said to be in a *mode*, and the interaction is *modal*. The interface is also said to be in a *modal loop* waiting for the user's response.

1.1 The Advantages of Conversation

The solution pursued in this thesis is to model more closely the interactions that people use when they communicate with one another—conversation. When a person talks to someone else, he may frequently interrupt a topic of conversation and move on to another topic. If the participants resume an old topic, they would like to continue where they left off. A command-driven system offers little means for the interaction to develop a shared context among its participants, imposing an unnecessarily rigid structure to the exchange.

Imagine a scenario in which a user can interact in a relatively natural dialog with a computer about his mail and their authors (C is the computer agent; H is the human):

...
C: This message is from Barry. Do you want to hear it?
H: Where is he?
C: He's in Lisa's office.
C: Call him?
H: Read it.
C: *reads Barry's message*
C: The next message is from Don.
C: Read it?
C: *waits for H's answer...*
C: You just received a voice message from Barry.
C: Play it?
H: Yes.
C: *plays Barry's message*
H: Where's Barry?
C: He's in the sound studio.
...
User converses with the agent one hour later.
C: Barry sent you a message while you were away.
C: Play it?
H: OK.
C: *plays Barry's message*
...

Figure 1.1: A model dialog.

The interaction is fluid, efficient and dynamic because the interface can keep up with the user's commands and questions at any time. Such interfaces can remove some of the rigidity in a speech interface by allowing the user to jump from point to point in the conversation, going in and out of task contexts and later resuming at the point of departure if desired.

1.2 The Advantages of Learning

Good conversation cannot occur if the computer is not a good conversationalist. The interface must become more active in its behavior; instead of the user always telling a program what to do, the program can make suggestions to the user about what can be done. Interfaces assume more active roles by suggesting appropriate information and actions to users based on learned experiences with the user. They can guide the user through the interaction, instead of simply awaiting the user's next command.

Computers also need to be better listeners and learners if they are to be better conversationalists. The suggestions they make can be a result of an inference process the agent makes about the user's goals and interests. These inferences can be used to track changing interests for particular kinds of information, and the user can be alerted to it as soon as it arrives or in a later session. In the above scenario, the agent has inferred that the user was interested in reaching Barry and alerted him to information of interest as soon as he conversed again with the agent.

1.3 Research Challenges

This thesis explores the issues involved in building a conversational agent in the speech domain. The new salient capability of such interfaces will be their dialog modeling and its suggestive behavior, based on machine learning algorithms. The research focuses on an application called Chatter, which is designed to facilitate communication in a work group. To this end, Chatter allows users to perform multiple communication-oriented tasks, such as reading message sent by other users, finding information about people, locating them, and calling people on the phone. The interaction is characterized as an agent which tracks the user-computer discourse and make inferences about the user's intentions. The former lets the interaction occur more naturally. The latter enables the agent to find and alert the user to useful information and actions.

Chatter is a telephone-based application that offers users remote access to personal desktop information, such as e-mail, voice mail and rolodex information. Input and output is provided through a telephone interface using a recognizer for input and text-to-speech synthesis for output.

Speech provides a challenging yet rewarding domain in which to study conversational interfaces. First, it is possible to leverage the strong implicit rules governing conversational that people have. If it is possible to mimic, and not violate, some of these implicit rules in an interface, computers can become more efficient and natural to use by voice. Second, speech

is already an established medium for remote communication and information. There already exists a wide array of media—telephone, e-mail, voice mail—with its complement of services and information available through audio access. More and more information will become available by voice, so the need for integrating these disparate sources and controlling them in a easy way will become important.

1.4 Overview of this Document

Chapter 2 analyzes the motivations surrounding the design of Chatter. It discusses the strengths and weaknesses of speech interfaces and argues that discourse modeling and machine learning may facilitate the use of speech interfaces.

Chapter 3 provides background research and applications related to Chatter. This chapter reviews research from several areas, including discourse, conversational interfaces and agents.

Chapter 4 describes an architecture for performing interaction over mixed mediums. It describes the implementation of an event server called *Zeus*.

Chapter 5 begins the discussion of Chatter's implementation by describing how speech input is processed into a semantic representation. It also describes the advantages of using both speech and buttons in a speech interface.

Chapter 6 outlines the discourse modeling aspect of Chatter. It reviews the Grosz/Sidner theory and explains Chatter's implementation. This chapter also addresses the interaction issues, including those of choosing segment boundaries, resolving pronoun references, terminating a segment, asking questions, reintroducing segments, generating responses, and repairing errors.

Chapter 7 addresses the agent and machine learning aspects of Chatter. It introduces memory-based reasoning and describes how it is integrated with the interface. The major issues it addresses are: how it is used to decide on interesting mail, suggesting courses of action, feedback to learning algorithms and integration of learning and discourse.

Chapter 8 talks about issues involved in using recognition in the context of a speech discourse system. It describes the design of Chatter's vocabulary and a recognition server.

Chapter 9 offers some conclusions and possible future work.

The appendices provide further technical detail about the implementation. Appendix A gives the Application Program Interface (API) for the recognition server. Appendix B

explains how to interface with the event server so that other applications may be written for it, and Appendix C shows the object-oriented design hierarchies.

1.5 Document Conventions

The conventions used in this thesis for distinguishing user speech and interface feedback are straightforward and hopefully decipherable from its context. Except in figures, user and machine utterances such as *hi, who's this?* are italicized instead of quoted. Actual implementation constants or code like the string `"%who"` are documented in Courier font. Finally, data structures and object classes are bold, such as **MBRStore**.

2. Designing Chatter

This chapter describes the considerations and motivations behind the design of the Chatter interface. It considers the tasks for which Chatter is intended, describes its functionality, and motivates the choice of discourse model and learning agent techniques to address some of the issues inherent with speech.

2.1 Task Domain

Chatter is an agent for facilitating communication in a work group. It is designed to be a front-end for a group of users who send and receive e-mail and voice mail, and who seek information and whereabouts about people with whom they collaborate. The scenario being considered is one where a group of users are working in several places, at the office or home and even during travel. As available communication devices become more portable and ubiquitous, the task of reaching people gets more complex because the means are more varied. People's work habits become mobile, and the steps involved in getting information to intended recipients are less direct. This mobility renders the once easy task of communicating with group members more troublesome.

Computers can be used to tie the communication efforts of all remote users together, through either real-time synchronous or store-and-forward asynchronous communication. Chatter addresses the following challenge: how does such a group of users keep in touch with one another when they are in remote places? How can users asynchronously deliver messages to one another from remote locations? How can remote users stay in touch with important events at the office? How can computers facilitate the desire of some users who want to switch from asynchronous communication to synchronous? This thesis seeks to answer these questions by providing a system which facilitates these goals.

To facilitate the exchange of information in a work group, the Chatter application provides a remote interface for staying in touch with the office when one is away. Because the telephone is a ubiquitous device, it can be used effectively as the peripheral for supporting communication.

2.2 Chatter Functionality

When people become mobile, communication usually means messaging, the ability to store and forward messages for other users they can read when it is convenient for them. While

communications becomes mostly asynchronous in this scenario, some types of tasks can be performed more effectively with synchronous communication or conversation. Chatter is designed to facilitate both synchronous and asynchronous means of communication by offering the following basic functionality:

- **E-mail reading.** Often, work is accomplished by passing e-mail messages back and forth, so the ability to read messages is almost critical for staying in contact. Chatter allows the user to play text and multimedia mail messages and to reply to them by sending voice messages.[1] Messages can be formatted as voice mail or multimedia mail with a voice attachment. Facilities for maintaining the user's mailbox is also provided; the user can save messages to temporary files for later viewing, delete or forward them to another user in the group.

- **Voice mail playing.** Voice mail is even more important for interfacing to the outside world, so Chatter has the ability to play a user's voice mail him through the interface. Voice mail messages are automatically maintained as digital audio files on the workstation and also have associated time of receipt and caller ID [Sti91], so it is possible to call a message's sender if the caller ID is available. Like the e-mail reader, the user has the ability to play, reply, delete and forward voice mail messages.

- **Phone dialing.** With a telephone as the interface, the user would expect to be able to place calls to people. How can Chatter facilitate this task? One of the common difficulties in reaching someone is having the number handy. Chatter can use telephone numbers found in the user's rolodex to place calls for the user. When the user makes a request to call someone, Chatter initiates a conference call, dials the number and patches the third party into the call. Chatter waits a brief period before terminating its participation so that the participants can converse. The purpose of the waiting period is to enable the user to cancel the call, say, because the called party does not answer.

- **Rolodex access.** To further facilitate the communication process, Chatter allows the user to access the information in his rolodex. While the rolodex is maintained at the workstation, its data is still useful remotely. For example, the user may need to find someone's fax number while the recipient is unavailable. The rolodex includes information about people's home, work and e-mail addresses, home, work and fax numbers, and any remarks about the user. These fields are all accessible from Chatter.

- **Activity information.** To facilitate real-time communication, Chatter can also inform the caller on the whereabouts of other's in the work group. This information is useful in

1. For users without the ability to receive multimedia mail messages, a facility for typing text replies using TouchTones is also available but is not incorporated due to time constraints.

case the user wants to know whether he can call someone at the office when it is more convenient to talk to him. Information about the location of users is collected from the group's set of active badges [Wan92], which people wear to receive notification of messages and to inform the system where others can find them. Information about workstation usage is also recorded. If people do not wear badges, it serves as a backup system for locating people. For example, the user may ask Chatter to locate the user `mullins`, and the system can respond with `mullins is in his office, active on monadnock` or `mullins was last seen about 45 minutes ago on whitney`.

Chatter borrows many of the capabilities from the group's Phoneshell application. The major objects of dialog in Chatter involves messages and people. The thesis is concerned about how this functionality can be fitted into the context of a dialog and how the interface can generate interesting suggestions of messages to read and commands to perform.

2.3 Chatter: A Sample Dialog

First, a script of a Chatter session is provided to demonstrate many of the ideas in the thesis. It is annotated to provide an account of the noteworthy aspects Chatter builds into the dialog. Note how the agent makes suggestions about both interesting messages to read and courses of action to perform. Also, note how the dialog weaves in and out of different topics, providing a natural flow to the user's interaction with the interface. In the script, recordings are given in italics:

C: Hi. Who's this? H: It's Eric. C: Hi Eric! You have two messages. Hear them? H: OK.	The user logs on to Chatter and identifies himself to the system. Chatter makes a suggestion about messages to read, and the user only has to answer with OK to go ahead or say something else he would like to do. What the user does first is remembered by the interface so it can be suggested later.
C: Message 1 is a voice message from Lisa. *Hi Eric, it's Lisa. I have the references you asked me about. I'm going to be at home today and may or may not get to Lab. If you need them in a hurry, give me a call. See ya.* End of message.	Chatter plays Lisa's recorded voice mail message for the user. The interface ranks messages from most to least interesting by observing how the user reads mail. Read messages are considered more interesting than those which are skipped over.

H: Where's Lisa? C: Lisa was last seen about ten minutes ago on Phoneshell. Call her?	The user interrupts the current topic of dialog to ask where Lisa is. Since Chatter has location information about users, it can give information about when Lisa was last seen. Chatter infers from past situations that the user is probably interested in calling the person when he asks this question, so it makes the suggestion.
H: OK. Call her at home. C: Calling her at home. *Conversation with Lisa takes place. Eric decides to go to Lisa's house to go pick up papers.*	The user affirms the suggestion and adds additional information to the command about calling her at home. Chatter uses the phone number stored in the user's rolodex to dial the number. Note the use of pronouns by in the user's utterance.
C: Anything else about Lisa? H: Yes, what's her home address?	After the call completes, Chatter reminds the user they've been talking about Lisa and asks if he would like to know anything else about her.
C: Her home address: 572 Main Street #118, Cambridge, MA 02139. H: OK.	Note how Chatter echoes the important information back to the user in the next utterance. Since speech recognition may often commit mistakes, the user at least has the opportunity to ask for the information again if the query was incorrectly heard. The address information is provided by accessing the user's rolodex. The user finally says *OK*, an acknowledgment that the topic about Lisa has completed.
C: Message 2 is from Don Jackson about "a phoneshell bug". Read it? H: OK. C: Eric, this message seems to be break phoneshell. Can you try to fix it? Thanks. ——original message from SPOOL@northeastern.edu follows: ...	Chatter returns to reading messages for the user. Unlike the previous message, Chatter first asks if this message is to read because it is potentially long.

C: End of message. Send a reply? H: No, forward it to Jordan.	The agent has noticed that the user usually replies to messages from Don, so it makes the suggestion to the user. The user turns down the request for another. This situation is recorded so that the agent can better predict the user's future actions.
C: Record your voice message to forward to Jordan. H: *Jordan, this is Eric. Can you see if you can fix this bug? Don is having some trouble with it. Thanks.* C: Sending your message.	The user records a voice message for Jordan. Chatter knows that Jordan can receive voice messages, so it decides that it would be easiest for the user to record a voice message.
H: Call. C: Call Don? H: Yes. C: Calling Don at work.	The user says call, an ambiguous command because no person is specified. Because of context provided by the message being read, the agent makes a guess as to who it may be.
No answer from Don. H: Cancel. Send a voice message. C: Send a voice message to Don? H: Yes. C: OK. Record your message. H: *Hi Don, it's Eric. I forwarded your bug report to Jordan, so it should be fixed pretty soon. Let us know if there are any other bugs you encounter. Thanks.*	Don is not at his phone, so the user exits from the call and decides to send a message. Again, since the send command has been completely specified, context is used to verify the recipient.
C: Sending your message. No more messages to read. OK. What now? H: Good-bye. C: See you later.	All messages are read, so Chatter has run out of topics to talk about. It asks the user what he would like to do, and the user signs off.
An hour later:	
C: Hi. Who's this? H: Eric C: Hi Eric! Don left a reply to your message. Hear it? H: OK. C: Eric, thanks! Don. C: No more messages. OK. What now? H: Good-bye.	The next time the user logs in, Don has left an e-mail message for him. Chatter considers this message to be important because the user has tried to call Don and was unsuccessful, so this message may be important.

Figure 2.1: An example dialog in Chatter.

2.4 Design Motivations

The dialog above shows how Chatter operates in a fairly natural dialog with the user. Not unexpectedly, spoken dialog is shown to be a meaningful medium for human-computer interaction because it is very expressive. Chalfonte, Fish and Kraut found that a paper revision task could be done more expressively if subjects communicated in speech instead of only being limited to marking up a paper draft [Cha91]. Subjects offered the most expressive, high-level comments for the task if they interacted even by one-way speech, while video and/or text mark-up only resulted in less expressive comments such as ones about diction and grammar. In their study, the addition of a visual display did not add much to the expressiveness of the subject's comments. The implication for speech-only interfaces is that they have the potential to be even more expressive than even visual interfaces in some cases, given that a computer can handle the expressive input.

2.4.1 Using discourse and speech

It seems clear that dialog is expressive because many conventions are already built in to make it faster. Many thoughts can be said in little time because participants already assume the conventions that others will follow. One of the most these conventions is the existence of a focus, a topic of conversation. The focus shifts from one topic to another during a conversation, providing a structure to the conversation. The focus establishes the object under discussion, upon which participants have come to agree. Unlike interfaces which do not keep history, conversations are efficient because participants maintain state, and they can use expressions such as pronouns to refer to that state without having to repeat everything. Unlike moded interactions, topics of conversations can also be interrupted; the speaker may decide to change or interrupt the current topic to establish another, while at some later point, he may want to reintroduce a previous topic and continue where he left off. If an interface can follow these conventions, the hope is that human-computer interaction can be faster.

The ubiquity of speech devices, such as telephones, also makes it a convenient medium to use. In portable situations, when the user's hands or eyes are busy, a modality such as speech allows users to operate the application without requiring them to switch their visual attention to operate the interface by physical controls. Conversation is also a natural means of interaction for people in the Chatter domain. Leaving and sending messages for people already encourages the user to converse, so the interface becomes more natural if it can be controlled through channels already being used.

Designing a good conversation requires making the computer's feedback brief and inter-

ruptible in order to conserve time and reduce the amount of information that the user must remember [Rud91] [Wat84]. In studying audio communication over telephones, researchers found that speakers alternate turns frequently, and utterances are typically either very short, or five to ten seconds long [Rut87] [Wil77].

It is also important for the user to have the appropriate mental model of Chatter. The danger of attributing too much understanding or intelligence to the computer once it can recognize speech is always present, for the user might conclude that the computer will now understand anything he says. As Negroponte notes about agents, "...And remember, we are not talking about discussing Thomas Mann, but delegating, which more or less means: issuing commands, asking questions, and passing judgments; more like drill sergeant talk." [Neg90] So, drill sergeant talk will do for now.

2.4.2 Using agency

Using conversational techniques for interacting in speech seems reasonable, but they will not work well if the dialog is always one-sided. If the user has to speak all commands, then, with the exception of the novice, users will be slower performing tasks speaking than using buttons. The conversation must be supported by a "desire" on the part of the interface to make useful conversation. In the realm of tasks, useful conversation means suggesting relevant courses of action. As Seneff, Hirschman and Zue note, "At one extreme would be a system that answers user's questions without at any point asking questions or making suggestions. At the other extreme would be a menu system that forces the user into a very limited range of choices" [Sen91]. They argue that the middle ground of the spectrum is the interesting aspect of spoken language systems. A system can take the initiative to reduce the time for spoken input by phrasing suggestions as questions, and the user has only to acknowledge the recommendation to proceed. Since individuals have varying habits in the domain, this motivation suggests the use of agency, a facility for learning about the user so that customized suggestions can be made. Since users can also have changing preferences over time, a learning agent can also adapt to them more easily than a strictly rule-based agent.

The concept of an agent is a powerful one. As Laurel states, "An interface agent can be defined as a character, enacted by the computer, who acts on behalf of the user in a virtual environment. Interface agents draw their strength from the naturalness of the living-organism metaphor in terms of both cognitive accessibility and communication style" [Lau90]. As with real-life agents, the user ascribes certain sensibilities to a program if it is called an agent. The metaphor Chatter seeks to present is one of an electronic assistant who tells the user about messages left for her since he last checked in. Since the assistant has chatted with

her before, it knows about her habits and style for managing messages. The user therefore expects the competent agent to make the right suggestions, and the computer can do so because it is expected by the user.

Maes also makes an important point about learning agents, as opposed to those who are fully competent right away. A learning agent should build up a trust relationship between it and the user. It is not a good idea to give a user an agent that is from the beginning already opinionated, albeit qualified. Schneiderman argues that such an agent would leave the user with a feeling of loss of control and understanding [Mye91]. On the other hand, if the agent gradually develops its abilities—as in the case of the learning approach taken with Chatter—the user will be given time to gradually develop a model of how the agent makes decisions. Initially, the agent is not very helpful as it knows little about the user's preferences. Some amount of time will go by before the assistant becomes familiar with his habits, preferences and particular work methods. Yet, with each experience, the agent learns, and gradually more tasks that were initially performed by the person directly, can be made more efficient by suggestions.

A hazard of calling an interface an agent may be that the user may ascribe too much competence to it and expect it to do or learn more than it is capable of. However, as in real life, assistants are not expected to know everything about their clients (although as much as possible certainly helps). Agents do not have to be omnipotent or omniscient beings but may have personalities that exhibit particular strengths and weaknesses, and any program will have strengths and weaknesses depending on how it is designed.

2.4.3 Building a portable communication base

Interface issues aside for a moment, as computing becomes more ubiquitous, the need for providing infrastructure to support interaction at any place and time increases. Instead of having one desktop terminal from which to access the information, there are now several interaction devices. Such a distributed system raises the issue of collecting and disseminating information in a consistent fashion. While the desktop interface has enough computing power to be smart about the user's preferences, can a small pager device also exhibit some of this same (or even simply consistent) intelligence? A related and interesting challenge is how to provide a channel for collecting training data for the agent, given that interaction techniques among the different media are different. This thesis hopes to address some of these issues as well.

2.5 Overview of Chatter

Chatter is designed with the above interface design motivations in mind. To try out these ideas, it is important to build a real, workable interface to receive the user's input and generate output. This thesis also describes the subsystems constructed through which the agent can communicate with the user. Figure 2.2 presents the high-level architecture of Chatter, showing the basic interaction loop of the system:

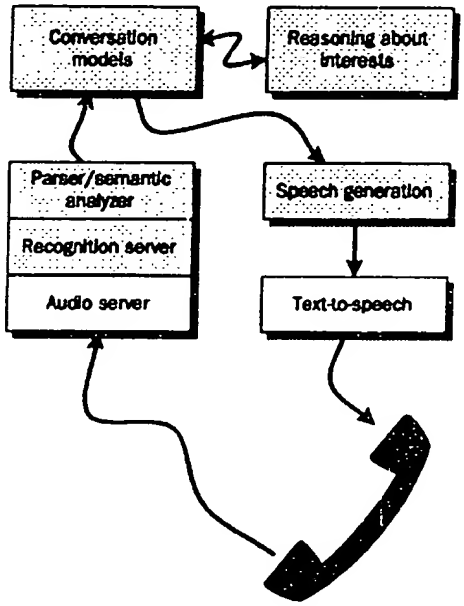

Figure 2.2: Overview of Chatter.

The shaded parts of the figure are addressed in the thesis. Information flows through the system in the following way. First, telephone audio is captured on the workstation through an audio server process. A recognition server receives audio from the server, and speech is converted into text. Then, parsing and semantic analysis are performed on the data to obtain a data structure representing the user's utterance. Processing takes place in the conversation module and interest reasoner, which share information about what is interesting and how to present it. The conversational model provides the context for the reasoner, which generates appropriate responses for the user. The response is given syntactic form by the speech generator and is finally synthesized for output to telephone. A DECtalk, a device for converting text to speech, is used to generate speech for output.

3. Background

Some of the groundwork is already set for the construction of conversational systems like Chatter. This chapter outlines the work preceding Chatter and begins to address some of the challenges in building conversational interfaces. The research for this thesis meets at the crossroad of several areas, including computational linguistics, system design and artificial intelligence.

3.1 Discourse

Perhaps the most necessary requirement for building conversational interfaces is to first come up with models of everyday conversation. Much work has already been performed in discourse theory, which studies and formalizes the conventions people use in conversing with each other. By following the conventions found in human-to-human discourse, computers can communicate with humans in a more natural way. Unfortunately, this field is too broad to describe in a chapter, so the topic will only be addressed briefly.

The most well-known theory in discourse is the Grosz/Sidner model, whose major contribution is that conversation is organized around some kind of stack. [Gro86]. Linguistic utterances in a conversation can be considered as means to achieving goals. Goals may have sub-goals, so discourse assumes a recursive structure that is recorded on this stack. As soon as the main purpose of the conversation is satisfied, the dialog completes. The theory provides a framework for investigating other aspects of discourse, including planning and anaphoric resolution. In Chapter 7, we go into greater detail about this theory, which we use to model discourse in Chatter. Recently, the field has extended these basic ideas to model more closely language acts—committing, accepting and negotiating—and their effects on the internal states of conversational participants [Sid92].

3.1.1 Plan recognition

Essential to figuring out suggestions to give to the user is figuring out what the user is trying to accomplish. Determining the user's intentions from his utterances is a major area of research called plan recognition. The basic approach to modeling intentions has been the embedding of a set of knowledge dependency trees into a natural language system. These trees track the state of the user's knowledge and discourse, allowing the system to maintain a set of possible high-level goals that a user may have at any one point during the exchange. When the computer wants to know what the user wants to do (perhaps to assist him), the

computer generates a query based on its set of hypotheses of what the user wants. Kautz gives a good overview of the issues and approaches of plan recognition in the last decade in [Kau90].

Many of these systems are confined to text-based interfaces, where either the user or an experimenter's have keyed the input, from which natural language processing occurs.

3.2 Conversational Interfaces

In contrast to text-based natural language interfaces, several other spoken language systems have already been built. Spoken systems, perhaps with some visual add-on, dominate the work that has been completed in the area of conversational interfaces, although some have applied discourse theory to multi-modal communication [Wah92]. This limited research has concentrated on the coordination of hand and eye gestures in disambiguating descriptions and anaphora in verbal input. Like Chatter, the following systems attempt to capture an interaction in a specific domain.

3.2.1 Conversational Desktop

The most demonstrable conversational interface to date is the Conversational Desktop [Sch84]. The Conversational Desktop provides an integrated interface for accessing voice mail, rolodex, calendar and other workstation-accessible information. Noteworthy is the dialog repair mechanism built into its grammar, which allows it to accept and repair partially recognized utterances by generating questions to prompt the user for the unrecognized input [Sch86]. The Conversational Desktop employs multiple modalities of interaction, combining both visual and audio interfaces for input and output of information. The environment suggests courses of action using hard-coded rules and information about what the user is doing (for instance, who he is speaking to on the phone to make decisions). It is a proof-of-concept system that makes only conservative use of past interactions to guide future behavior. Like Chatter's task domain, the Conversational Desktop is designed to facilitate work group communication.

A precursor to this conversational interface was Phone Slave [Sch85]. It used dialog techniques to take a telephone message, asking callers a series of questions and then recording the answers in separate audio segments. Although no speech recognition was used, structure was provided in the audio recordings, and the segments could be assumed to correspond to the answers of the recorded questions.

3.2.2 Voyager

Voyager is an attempt to use a more sophisticated discourse model with speech recognition [Zue89] [Zue90]. It is an urban exploration system which can answer questions about landmarks and roads in Cambridge between MIT and Harvard University. It has a speech recognition front-end and generates output in the form of a map, written answers and spoken output. Some things one may ask the system are *do you know the most direct route to Broadway Avenue from here* or *can you tell me how to get to the Chinese restaurant?* If a new command is not completely specified (perhaps the destination is ambiguous or the source is unknown), the system generates questions to obtain the missing information. Voyager uses a simple discourse model for pronominal resolution using two slots which refer to the most recently referenced set of objects. It also maintains a stack of incompletely specified commands, which gets popped off and executed when new information completes the ambiguous commands.

In their papers, Voyager's designers drew several significant conclusions about speech systems: first, speech recognition must be integrated with natural language in order to achieve speech understanding. Second, systems must be designed to support realistic application domains and be able to translate spoken input into appropriate actions. Third, systems must begin to deal with spontaneous speech, since people do not always speak grammatically well-formed sentences when conversing. The last point highlights the importance of studying natural language processing in the context of spoken speech.

3.2.3 MIT ATIS

Recently, more ambitious attempts are being made to study discourse in the context of speech. The MIT ATIS system, or Air Travel Information System, answers questions about airplane flights for a traveler interested in booking a flight [Sen91].[1] Its database consists of tables from the Official Airline Guide containing information about destinations, flight numbers, fares, times, connections, seat availability and meals. One of the most interesting aspects of the MIT system is the design of its initiative-taking interface. The system can be operated in both "non-booking" and booking modes. In the former, the user can ask questions and get responses. In the latter, the computer takes the initiative by engaging in a goal-directed dialog with the user, carrying him through the booking of a round- trip ticket. The system asks questions in order to fill in an on-screen facsimile of a plane ticket, and the user can negotiate with the system to find the best available package. This ticket is incrementally filled in as the information is given and provides visual feedback that the information has

1. ATIS is a DARPA-funded program aimed at developing guidelines for evaluating spoken language systems.

been accepted. The two modes of interaction are switched manually.

Seneff, Hirschman and Zue described the lessons they learned about the importance of discourse systems:

- A system should permit the user to build up a solution incrementally and therefore make implicit and explicit reference to information from earlier parts of the dialog.
- In certain domains, it is natural to have the system play an active role in the dialog. Spoken language programs will never reach a practical level of performance unless attention is paid to issues of discourse and dialog.

3.3 Audio-Only Interfaces

Some researchers have recently begun to explore the interface issues surrounding audio-only interfaces. The main challenge in this domain is that information feedback and user feedback must be entirely auditory. Unlike graphical displays where lots of information can be organized and displayed simultaneously, the auditory domain is linear and its bandwidth must more limited. New methods must be developed so that information and feedback can be presented coherently. Chatter is an audio-only interface that allows user to scan through a list of messages, so scanning techniques must be used to allow access efficiently. The following projects explore how user feedback can be given for navigating information spaces. These examples also demonstrate that speech and audio can be made more useful when structure is given to audio.

3.3.1 Hyperphone

Hyperphone is an audio-only hypermedia system for exploring voice documents [Mul90]. Voice documents are composed of nodes (small fragments of text) and links connecting nodes. A linear sequence of nodes is called a Parsed Voice String (PVS), whose purpose is to organize collections of fine-grained text objects to provide a linear presentation and also allows for interruption and "excursions" to related subtopics. Links can be literal or virtual. A literal link is a semantic connection between two pieces of information while a virtual link is a temporal connection, which is created dynamically as a result of user interaction. An example of a virtual link is the most recently traversed node. This allows the user to issue a query such as *where were we*?

The user interface for Hyperphone is speech-only—navigation through a voice document is controlled by voice input and the output is generated using text-to-speech synthesis. Navigational commands are provided by a speaker-independent recognition vocabulary. Since

the recognition lacked many of the necessary technical and domain-specific words, the authors propose the use of "voice direct manipulation" to handle navigation of the database. For example, a voice menu would be presented and the user could select an item by saying *that one* or *do it* when the desired item is spoken. Muller and Daniel, the authors of Hyperphone, found that very fine-grained objects are needed in order to provide a conversational dialog instead of a fixed menu structure for navigation. Fine-grained segmentation of the information allows queries such as *tell me more about this topic*, *skip the details*, and topic-specific queries such as *who wrote the paper you're describing?* They also emphasized the importance of interruptibility because of the sequential rather than simultaneous nature of the speech interaction.

3.3.2 Hyperspeech

Hyperspeech is another hypermedia system [Aro91] similar to Hyperphone. Hyperspeech provides the ability to navigate through a network of digitally recorded segments of speech using isolated-word speech recognition. Information is presented using digitized speech output as opposed to synthetic speech. The recorded information is spontaneous speech from interviews with five interface designers on the topic of the future of the human-computer interface. Responses to questions were manually segmented into nodes, and links were added between related nodes.

Users can navigate through the database using a variety of link types. For example, a name link allows the user to jump to comments made by a particular speaker related to the current topic. Another link type called a control link provides navigational commands such as *browse*, *scan*, *more* and *continue*. *Browse* makes a transition to the next summary topic node, and *more* makes a transition to a detail node in the current topic. This navigational concept has recently been extended by others to browse a series of news articles captured from radio. The news browser is called Hypernews [Mul93].

Arons, the author of Hyperspeech, concluded from this project that: speech interactions should be streamlined to provide concise yet informative feedback; system actions should be interruptible and the systems' response to the interruption should be timely; a flat recognition vocabulary, where all commands are always active, takes advantage of the goal-directed nature of speech.

3.3.3 VoiceNotes

Unlike the previous two projects, VoiceNotes enables the collection and retrieval of user-authored audio data. VoiceNotes is an application for a voice-controlled hand-held com-

puter which allows users to create, manage and retrieve user-authored notes stored as voice [Sti92] [Sti93]. These notes are small segments of digitized speech containing thoughts, ideas, reminders or things to do. VoiceNotes explores the problem of capturing and retrieving spontaneous ideas, the use of speech as data, and the use of speech input and output in the user interface for a hand-held computer without a visual display.

In VoiceNotes, recorded speech is stored in segments called voice notes, and such notes can be organized into lists. The interesting aspect of VoiceNotes is that audio can be arbitrarily structured by the user. New categories can be dynamically created and note management can be performed completely using speech. Later, notes can be retrieved by saying the name of the list under which they are stored. New work on VoiceNotes allows other attributes besides list membership to be attached to notes, so that they can be used to cross-reference notes across several lists.

Among other significant conclusions, the author emphasized the importance of providing a modeless interface that can more closely follow the non-linear interactions humans have when using speech interfaces. They expect to be able to change the context of interaction easily, without the system imposing unexpected contextual interpretation to user commands. She also noted that the use of speech did not displace the need for physical controls, such as buttons, for navigating in fine-grained information.

3.4 Computers and Telephony

Work in speech-only interfaces has also been performed in the context of telephony. Since telephony encourages conversation, it is a natural arena for exploring applications using voice. Interesting research examples include Etherphone from Xerox PARC [Zel88], MICE from Bellcore [Her87], PX from BNR [Kam90] and Phonetool from MIT and USC-ISI [Sch89]. Such applications can allow users to place calls from on-line address books: capture calling party information to assist with replying to voice mail, and even route calls based on who is calling, the time of day and other conditions [Won91]. Chatter's interface is used over a telephone, bringing unique design considerations as a device with a predefined set of button controls. The following systems have explored how TouchTones can be used as interaction controls.

3.4.1 IVR systems

The largest group of telephony applications are so-called Interactive Voice Response systems. These are usually commercial programs used mostly by the general public for accessing up-to-date information, and applications include operator assistance, bank account

queries, movie listings, and telephone directory assistance. Such systems use voice as output and TouchTones for input. Recently, limited speech recognition has been used to make input more natural. Yet, recognition is used little more than as "speech buttons," where a spoken utterance might contain a keyword which directly corresponds to some keypad press. These applications are also geared toward a wide range of users, so the functionality of such interfaces is usually designed to be simple and limited.

3.4.2 Phoneshell

Revisiting some of the challenges in the Conversational Desktop project, the Speech Research Group at the MIT Media Laboratory has recently created an IVR system geared to the more experienced user, in an attempt to study issues raised by the need to present more complex information in the audio domain [Sch93] [Sch90] [Sti91]. Phoneshell is a telephone-based interface to personal desktop information such as e-mail and calendar. The application allows the user to dial in to access workstation information through a suite of applications and menu choices. It is an attempt to compensate for, and possibly duplicate, the informational expressiveness of graphical media in the audio medium. It attempts to accomplish these aims by filtering and summarizing information before presenting it to the user. The command structure is very much conventional, however, and is completely user driven.

Phoneshell is an system which allows users to access voice mail, rolodex, calendar, dial-by-name, e-mail and pager management applications stored at a workstation. Users log into the system via telephone and command it via TouchTones. The system gives verbal response to the commands given by the user via synthesized speech generated by a DECtalk. The menu structure of the program is rather conventional, consisting of a hierarchy of menu choices navigated through the use of telephone keys. Upon entry to the system, Phoneshell presents a top-level menu for choosing any one the six applications mentioned above. These choices are presented sequentially—it announces, *for voice mail, press 1; for your rolodex, press 2;* ... When the user chooses an application by pressing a TouchTone, Phoneshell activates the application requested. The interaction is then rather moded; all subsequent interactions with the system are limited to the domain of the active application, and users cannot jump from application to application without first exiting the current application. For instance, if the user wants to listen to new messages, he hits a key to start the playback of the first message. To listen to the next message, he may either hit another key to play it or wait until the first message has finished play and a time-out period has occurred. To activate another application, the user must exit the current application by pressing a key to return to the main menu and then entering another application.

The following section will briefly describe the functionality of each application within Phoneshell.

- **Voice mail.** This application allows the user to read, send and maintain a mailbox of voice mail messages. Voice mail messages for the group are stored on the workstation as digital sound files, and this application allows access to those files. The application lets the user play newly-received message, listen to old messages, send or forward messages to other voice mail subscribers, delete unwanted messages, and record personal voice memos that appear only on the workstation screen.

- **Rolodex.** The talking rolodex, called Rolotalk, can be used to find information about people kept in a personal rolodex database. The user punches the first few letters of a person's first or last name and Rolotalk finds the card(s) matching the letters. Any field of the card can then be recited to the user through a choice of options. Since an e-mail address field is kept on each card, the user can send a voice message to the person within the application. Also noteworthy is the ability to use the phone number fields on each card to directly place a call to the person. This feature obviates the need to memorize phone numbers.

- **Calendar.** The application lets the user find out what appointments and reminders he has on his calendar. The application not only gives day views but also provides week-at-a-glance and month-at-a-glance views that give high-level summaries of the information on the calendar. This calendar is unique because audio segments can be recorded or transferred from voice mail into the calendar to serve as reminders. In addition, a user can request that his calendar be faxed to him by specifying a fax phone number. A description of its design appears in [Sch90].

- **Dial-by-name.** Dial-by-name is a rather standard implementation of the dial-by-name service found in many IVR systems. A user can enter a voice mail subscriber's name and the system will place a call to the recipient.

- **Electronic mail.** The user can use this application to read e-mail messages in his mailbox. The user not only can read his messages but can also reply to or send new messages by either recording a voice message, which becomes encoded in one of the many available multimedia mail types, or use the telephone keypad to type in a text message. The mail reader uses a filtering scheme based on a regular expression search that allows it to present more important messages first, enabling the user to get to the more relevant messages quickly. Similar to the calendar, the user can also request that lengthy messages be faxed to him.

- **Information.** A facility is included for accessing information about traffic and weather

and listening to news broadcasts from several sources.

Phoneshell currently runs on Sun workstations and is available for all members of the group to use. The system has been used regularly by several members of the group, who use it on a daily basis to access their messages when away from the office.

3.5 Agents

Once machines can speak and be spoken to, it seems inevitable that people will begin to impart them with human-like personas, so researchers have thought that computer systems can be made more realistic by actually giving them personalities. The idea of using agents in the interface to which tasks can be delegated was introduced by Negroponte [Neg90] and Kay [Kay90]. Agents are distinguished from other interfaces by their ability to learn the user's habits and adapt to them over time. Several computer makers have touted the idea of agents to illustrate their vision of the ultimate interface. Two of the most visible are Apple Computer Inc.'s Knowledge Navigator and Japan MITI's FRIEND21 project. In these demonstration videos, the interface is personified as a agent with speech and visual representations. Yet, the most interesting aspect is that the user interacts with the agent mostly by speaking to it (or him or her), pointing to the naturalness of speech as an interaction medium. Even though much thought has gone into modeling and constructing agents, currently available techniques are still inadequate from being able to produce the high-level, human-like interactions depicted in these videos.

Recent studies indicate that agents used in the context of interactive speech interfaces result in more information-efficient interactions than non-interactive, one-way systems [Ovi92]. The reason is that users are able to assume the existence of some being, either real or fictitious, who can serve as a memory for objects and relationships presented in the past. The efficiency is also shown to stem from the greater use of pronouns in the conversational mode than in non-interactive or written modes, making it possible for utterances to be abbreviated. If elements of this natural efficiency in conversation can be captured in an agent, it may pave the way to more fluid, natural interfaces. Here, we outline some of the work in agent interfaces.

3.5.1 Oval

An elaborate example of manual customization is the Oval system, which, among other functions can, implement an agent for sorting, delivering and flagging e-mail as it arrives at the user's mailbox [Mal87]. In the system, an agent is viewed as a set of rules. Each rule is specified as a set of features—the sender, recipient, subject heading, and so forth—that can

be detected given a message. The actions of rules that match are automatically taken, which may include delivering the message to any one of a user's mailboxes or simply deleting the message.

Directly specifying has the advantage that the user can explicit tell the system what to do in given circumstances. This capability wields much power, particular if the user knows how to express these rules to the system. However, Oval, and other rule-based systems, exhibit several difficulties from the user perspective: first, a user must learn how to translate his thoughts into the rule language, which may be arcane. Second, a user's preferences and work habits may change over time, requiring him to maintain and evolve these rules. The difficulty is exacerbated by the fact that since the rules are difficult to interpret and reprogramming is only done on occasion, the user must re-learn the rule language time after time. Most significantly, as experience with rule-based systems such as [Won91] demonstrate, the behavior that the user wants his agent to exhibit cannot always be captured in a concise rule set; there are often exceptions to the rules that the user wants. These exceptions cause frustration to the user who wants his computer to take note of them.

3.5.2 UCEgo

Instead of having user-specified rules, UCEgo is an attempt to encode all the rules for a particular domain and then to deduce which set applies to the user [Chi92]. Chin built UCEgo to try to infer from a user's questions his higher-level motives. The UCEgo agent has a large knowledge base about how to use UNIX, incorporating goals and meta-goals and performs planning, for example volunteering information or correcting the user's misconceptions. In this system, an interface agent called UCEgo gives on-line natural language text help to computer users through the detection of a hierarchy of goals and meta-goals of both the user and system. It uses the awareness of these goals to drive the contents of help messages as the user follows a particular line of questions, adding more information to responses of questions asked. Because UCEgo is isolated from the rest of the system, it cannot actually execute commands on behalf of the user. Most of the dialog is constrained to telling the user what to do. This leads to several limitations, the most significant one being that it is not possible to incorporate the actual command execution in inference about the user's goals.

As Maes notes in [Mae93], a limitation of UCEgo is that it requires a large amount of work from a knowledge engineer; a large amount of application-specific and domain-specific knowledge needs to be encoded and little of this knowledge or the agent's control architecture can be used for building other agents. A second problem is that the knowledge is static, possibly incorrect, incomplete, not useful or even idiosyncratic to the designer. The knowl-

edge can neither adapted nor customized. Third, there is a question of whether it is possible to provide all the knowledge an agent needs to be able to make sense of the user's actions.

3.5.3 Learning interface agents

To solve the difficulties associated with hard-coded rules, Maes and Kozierok recently implemented visual learning agents for scheduling meetings among a group of users and for deciding a user's interests of incoming e-mail messages [Koz93] [Mae93] [Koz93a]. Each user has his own agent that can be customized to individual preferences and habits. In the case of the scheduling agent, a user who decides to schedule a meeting can either manually mark the group's calendar or ask the agent to suggest the best meeting time for the group. In the latter case, the originator's agent queries the agents for other users, asking them for feedback the goodness of the suggested times. Each user may refuse the request, select a new time or enable his agent to help negotiate for a better time. At first, the user has to manually choose the action to take when receiving requests, but as time passes, the agent's observations of its user allows it to recognize patterns in the user's behavior and eventually learn to predict the user's action. The agent can then tell the user what it thought he would do. When the user gains enough confidence with the agent, he can hand control over to it. A similar interaction technique is used for sorting e-mail messages. Here, a screen snapshot from the program shows the agent predicting what the user will do when a meeting time is suggested by another user:

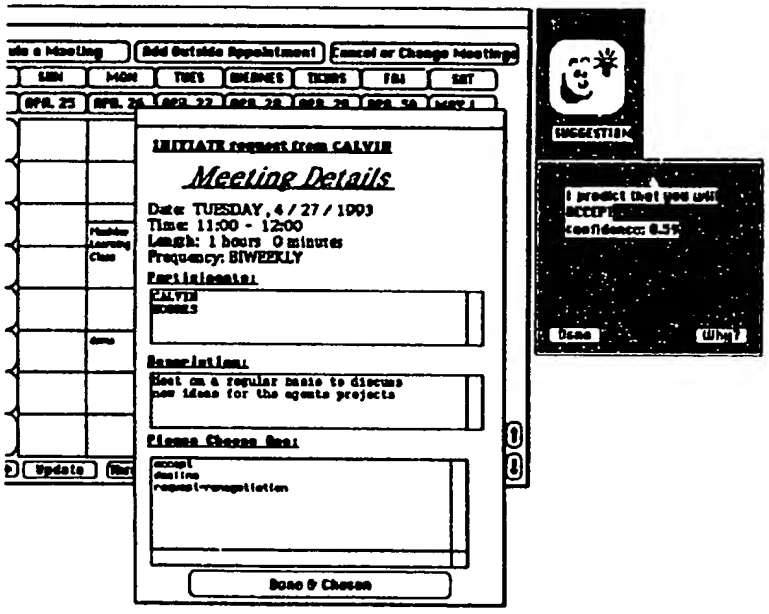

Figure 3.1: Meeting scheduling agent.[2]

2. Snapshot courtesy of Robyn Kozierok.

The agent records a history of all interactions and their results, using it as a basis for comparison to new situations. The authors use a memory-based reasoner [Sta86] to store past occurrences to infer appropriate actions for new situations, and a reinforcement learning algorithm to adjust weights of importance of keywords and proper names. When the agent is asked to schedule a meeting or new e-mail arrives, the agent inspects its database of examples and makes a prediction. Depending on how confident the agent is about its prediction, it either carries out the recommendation on the user's behalf or asks for confirmation.

Learning is accomplished either by passively observing the user in real and training situations or by actively asking for feedback when an incorrect prediction is made. In this case, the user is asked to weigh some of the factors used in making the decision. For the meeting scheduling task, these include the importance of the initiator of the meeting plus the importance of the schedules of other participants. In the e-mail task, factors include the importance of the sender, time of day, subject line, and other fields in a message's header.

3.6 Summary

Chatter in many ways combines and builds on research already performed in the areas discussed in this chapter. It extends the concept of the agent interface by combining ideas from discourse and machine learning, both of which are crucial in constructing agents in the speech domain:

- Even though spoken language systems such as Voyager and ATIS have been built using ideas from discourse theory, these systems so far have relatively simple models of discourse. Chatter represents a more ambitious attempt to model discourse more fully for use across several tasks and domains. This functionality entails the modeling of interruptions and reintroduction of topics in dialog. A secondary goal is to build a general framework or toolkit on which conversational systems for other domains can be constructed.

- While systems like UCEgo and Phoneshell allow the user access a mass amount of information, there is little facility for customization or personalization of the information. As spoken language systems become more conversational and personal, it seems desirable that they should listen and tune themselves to user preferences. VoiceNotes allows personalization to the point of arbitrary information structuring, yet the interface is still completely user-controlled. Chatter attempts to define a new role for interfaces by learning about user preferences and suggesting appropriate courses of action in the context of a dialog.

- Chatter also attempts to integrate some agent-oriented work, such as Oval and the work by Maes and Kozierok, and embodies them in the realm of speech interfaces. This project studies how interfaces can be designed so they can naturally collect information necessary for learning.

4. Building an Information Base

As computers become smaller and more ubiquitous, work patterns will become increasingly distributed as people are freed from the electronic shrines at their desks. Wireless networks, along with laptops, palmtops, pagers and portable phones, enable users to keep in touch with their information away from the office. In turn, the advent of ubiquitous computing has enabled computers to gather previously uncollectible data about people's daily activities for use by other users. Such a distributed system raises the issue of collecting and disseminating information in a consistent fashion; it becomes more challenging because the means of gathering and distributing information are much more varied than before. With remote interfaces such as Chatter, the information infrastructure needs to be rethought to make collection and dissemination workable in the distributed setting.

This chapter outlines the design and implementation of an information server called *Zeus*, an active process for collecting information from disparate sources and disseminating it in a consistent manner.

4.1 Motivations

In the Chatter domain, two types of information need to be collected for use by the interface: (1) static information whose contents do not change. A mail message is the key example of static information. They are objects that may be manipulated as a whole; they are read, deleted or forwarded indivisibly; (2) dynamic information about users. This information includes people's locations and their nearest phone number. In contrast, the contents of this information changes quickly.

While remote communication devices allows for new means through which to access information, they also introduce the need to maintain consistency among the different channels through which users can communicate. Abstractly, the user can be considered to have several interface devices through which he can access a database of information—either using a hand-held computer, a telephone or the workstation at his desk. The user would like the changes he makes at one terminal to be reflected at all other terminals. This way, he gets a globally consistent view of his information. Yet, such of a user's personal information is not stored in a form that conveniently allows for this degree of consistency.

Most of a user's information is stored in files in a filesystem, and it levies burdens on each of the application programs which try to maintain consistency with these databases. For

example, electronic mail is frequently stored as a flat file or a set of files. In the past, when mail was usually read in one place, this method sufficed as a means of storing the information. Today, it is possible and desirable to read mail using a remote interface. Since users will want to perform maintenance on their mailboxes at the same time, it is also desirable to have the desktop graphical mail-reader take note of these changes. To maintain consistency under the current scheme, participating programs not only have to be careful not to overwrite each other's changes during updates, they also have to update their in-memory store of the data with the changes other programs have made. Not only are there synchronization problems, there are also practical consequences of requiring each application to know how to retrieve information from such databases and to know how to reconcile the updates that have been made. As a result, programs become larger in size and run slower. With distributed computing, multiple processes are beginning to access personal databases that were only meant to be accessed by one process at a time.

One issue which complicates this picture is the response time of agent interfaces. Invariably, intelligent agents must perform some preference processing on the information for the user, and this processing usually takes a non-trivial amount of time. If the information changes, the agent will often need to recompute these results. If this operation isn't performed until it is actually required, such as when the user logs into the interface, then there may be a lengthy start-up time, for the agent must load the database, resolve the changes with its own in-memory database and finally perform the extra processing. If the computation can be done ahead of time, as soon as a change is made, then such interfaces will have a better chance to be ready for use when their users demand it.

On the information collection side, the kinds of information that are beginning to be gathered about users come from a variety of sources. This information usually comes in a raw form that must be processed before it becomes useful. For example, one of the most oft-discussed devices are active badges [Wan92]. These badges are tracked by a set of sensors spread out across an area. Information from each sensor needs to be pooled with information from other sensors to build a coherent picture of a person's location. Though the computation is straightforward, some processing must first be done before it becomes semantically meaningful to processes trying to use the information. The necessity of processing tends to hold for any distributed sensory system where there is an abundance of sometimes conflicting information. These include active badge networks, networks of computers, or networks of phones whose usage is being tracked.

The consequence of distributed data collection is that the means for acquiring such information becomes more complex, and it is reasonable to conclude that processes which use the information want it at a higher semantic level than sensors can directly provide.

These challenges call for an information architecture in which some intermediate ground is provided between information providers and users. The middle ground should provide a common-denominator representation of information which all processes can consider useful. Providers can deposit their processed information with this entity, from which information users can obtain information they can readily use. An information server can give this intermediate ground.

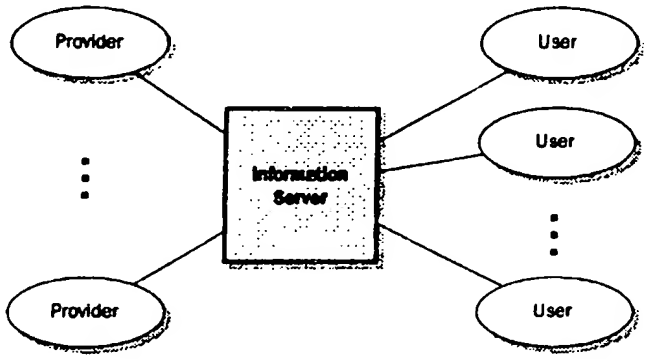

Figure 4.1: General information server architecture.

An additional benefit of this architecture is that providers do not have to know exactly who the users are or how to deliver the information. As Lövstrand argues [Löv91], separating the tasks of information generation and delivery gives several advantages; it relieves the producers of having to know how to deliver the goods to the appropriate recipients, the set of which may be changing and the means varied. For the receiver, this scheme alleviates them from having to know how to find the information, which may be coming from different sources. Lövstrand casts the argument in the context of a group of users, but the argument applies convincingly as well to a network of users and active agent processes accessing information. The server is a repository where information is recorded and used.

4.2 Related Work

An information server, also called an *event server*, has been implemented to store event information and electronic mail messages. Khronika is an event browsing and notification system developed at EuroPARC for calendar events [Löv91]. It is implemented as a shared network server with client processes, which add events to the server and receive automatic notifications of other user-preferred events. In Malone's Information Lens system [Mal87], an *Anyone* server serves as intermediary between senders and recipients of messages. Users send messages to the Anyone server rather than directly to individual users or mailing lists, and the server processes all potential recipients' rules and determines the actual recipients.

The unique aspect of an event server for the purposes of this interface is the diversity of information to store. Not only do we want to store calendar events and mail messages, but we also want to save location and activity data about people, or even the latest weather information. Any information that can be potentially accessed over multiple interfaces is a candidate for storage in the event server. At present, the kinds of information to store include electronic mail, voice mail, activity information about people, and phone calls people receive. We want a general scheme for representing the differing kinds of information. All of this information can potentially be used by an application such as Chatter, but clearly, the information is also useful for on-screen applications.

4.3 Design Criteria

The information server serves as a repository for information about people as well as other relevant entities. It should be simple in nature; the "smarts" for interpreting and using the events will reside in separate processes. We make the distinction between *client* and *daemon* processes. Clients are processes which deposit information into the server, and daemons are processes which use information in the server. The distinction is not critical because both access the server in the same way, and indeed a process can serve as both client and daemon to the information server. The central design issues for the information server called *Zeus* are:

- **Extensibility.** As new sources of information are found to be useful, users will want to add them to the pool of existing information. New event types should be able to be easily added to the database. The number of types ought to be dynamic, and adding new types to the server should not require any protocol extension of the entire server. This makes the server easily extensible for new types of events in the future and allows for fast prototyping of information representation.

- **Representational flexibility.** Data will likely come from different sources, and their structure and content will likely vary from one another. No fixed representation will adequately meet the diversity of all the information to be stored. A large class can probably be represented as records of key-value pairs, but the value fields may contain anything from a number to audio data. The structure of the information the server can store should be as flexible as possible, allowing clients which provide and use the data to choose the best representation for the information at hand. Again, this feature allows for fast prototyping of information representation.

- **Notification.** The ability of the server to notify clients of changes in information is a key feature of any server that purports to maintain information that changes often. This feature allows clients to be notified of changes as they happen so that processing can be

done, rather than requiring them to inefficiently poll for changes. *Zeus* makes it easy for clients to be notified of information updates in the server. Since the set of event types is not predefined, there can be arbitrary kinds of events in which clients can register interest, and the challenge lies in specifying a protocol which provides a simple way of registering interest and getting notified of changes independent of event type.

- Efficiency. The information server will likely serve a host of processes for a host of users, so it is desirable to have the server be as quick as possible in its transaction processing. Since it is a central store, it is easy to see that the server will become a bottleneck for processes if retrieving information takes time.

Zeus is designed to be flexible in the information that it is able to store. The server only makes minimal assumptions about the information contents and structure.

4.4 Server Design

This section describes the design of the server. It describes how information is structured in the server and the data representation used to store the information.

4.4.1 Information structure

To meet the above design goals, *Zeus* is constructed as a hierarchical database with notification capabilities attached to points in the record space. A hierarchy is used because it provides a convenient way of organizing groups of related information, and it is also fairly straightforward to implement. Information is organized around a set of *folders* and *records*. Folders represent parent nodes in the database tree and can store records as well as folders. The amount of folder nesting is arbitrary. Folders are a means for structuring the stored information. Three types of changes can occur to them: (1) new records or folders can be added to folders; (2) the contents of existing records or folders can be changed; (3) records or folders can be deleted from the folder. Records represent the child nodes and store the actual information. The form of this information is discussed in more detail below. Folders and records have unique identifying strings. For example, the folder which keeps information about users in the group is called `/user`. Within it are folders for users, such as `/user/mullins` and `/user/lisa`. The `/` character is used to delineate nesting level. The naming convention for folders and records in *Zeus* is similar to that used for the UNIX file system and was also inspired by FRAMER [Haa92], which uses the same naming scheme for frames.

Notification is performed on a per-folder basis. Processes which want notification are called *daemons*, and they inform servers of their interest by registering a callback function for a

given folder. Whenever the contents of a folder are changed, all registered processes for that folder are notified of the change. If the change was an addition or change, then the new contents are also passed along to receivers in case they may be interested. The notification mechanism is designed to be simple and straightforward.

The server protocol then simply allows clients to create, delete and change folders and records, and it also enables them to register for notification. The protocol is unencumbered; it does not actually specify what information is stored in the server but deals only with folders and records.

4.4.2 Data representation with dtypes

To make few assumptions about what information is stored or how it may be represented, we need a representation system that is fairly flexible. The questions of actual representation should be left to the information providers and users, who can best choose the representation appropriate to the task. We decided to use dtypes, a simple and powerful object-oriented representation scheme based on Lisp-like lists [Abr92]. dtypes store data inside objects. Atomic dtypes include integers, reals, strings and binary streams, and an aggregate list dtype allows for the combination of these, including lists inside lists. The flexibility of list dtypes, both in the kinds of data they can store and their variable length, is the key feature which gives the server its representational flexibility. This is the main reason dtypes were originally chosen to be the storage scheme. A specialization of a list dtype, called an *environment*, can be used to represent frames with key-value pairs. For example, an environment representing an e-mail message is:

```
(("date" "Sat Feb 27 22:40:52 1993" )
 ("from" "lisa@media.mit.edu" )
 ("name" "Lisa Stifelman" )
 ("to" "mullins@media.mit.edu" )
 ("subject" "mail filtering" )
 ("h-offset" 45345 )
 ("b-offset" 45944 )
 ("b-chars" 976 )
 ("rank" 2 )
 ("say" "important %s" )
)
```

Figure 4.2: Example dtype environment.

where the parentheses represent list nesting. The first string in each sub-list is the name of the field; the second is the value of that field. Environments have convenient access methods for storing and looking up keys and values. Since key-value frames are likely to be the most-often used representation of information, dtypes are also used for this purpose. dtype

objects can also read and write ASCII and binary representations, making them readily amenable to creating persistent databases. The representation of the actual events are described in more detail in "Organization of Events" on page 45. The use of dtypes allows the arbitrarily-structured information to be stored. The server also stores records as dtype objects, and clients can create dtypes and send and receive them from the server.

4.5 Server Implementation

The server is essentially implemented as a manager of dtype objects. Since dtypes can conveniently store other dtypes, the memory of the server is one large dtype holding all records. A corresponding dtype structure holds directory information about this memory. Besides storing directory names, this structure also holds attributes for each directory, such as the number of daemons who have registered for notification and how many clients are currently updating in it.

4.5.1 Buffered notification

One practical consideration of the notification scheme is the cost of communicating changes to processes. If a client wants to make a series of changes to a folder and there are several processes desiring notification of such changes, then every change will result in a wave of notification messages to each notifiable process, and the time for performing the set of changes and resulting updates will be longer due to network traffic overhead.

To solve this problem, a simple buffering scheme is implemented. A client which knows that it will make a series of changes can inform *Zeus* that it is about to do so. The server will accumulate the changes until it is told that all updates have completed, at which point it notifies daemons of all changes. Since multiple processes may want to update the same folder, an increment/decrement method is used. When a process notifies the server for update, the folder's update count is incremented. Similarly, a process which has finished updating results in decrementing the folder's update count. When the count is 0, daemons are then notified of the changes. Another caveat is that the buffer is not a simple history of changes. If during a buffered update, a record is deleted and then later recreated, a daemon can view the update of that record to be a *change*, not a deletion and a creation. This compilation in effect makes the update atomic, makes the bookkeeping for updates simpler and more efficient to transmit. A simple state machine is used to handle these transitions:

State \ Change	none	New	Delete	Change
New	New	n/a	none	Changed
Deleted	Deleted	Changed	n/a	n/a
Changed	Changed	n/a	Deleted	Changed

Figure 4.3: Buffering state changes on records. *n/a* means the transition is not allowed. *none* means the update information is discarded.

4.5.2 Data persistence

The information server is designed to run continuously from the clients' point-of-view. Indeed, one of the ideas for having such a server is to abstract away the storage details for clients. In theory, all server data can be stored in memory without the need to go to disk, since the information is relatively small, will be frequently accessed, and will be updated often. In practice, the machine on which the server runs will be restarted periodically, so the server should make some provisions for storage in a more permanent place.

To do this, *Zeus* implements a simple method for saving state. It maintains a timer, which periodically expires and instructs the server to dump its state to disk. The older contents of the file is entirely replaced, so that the file contains only the latest state of the server. When *Zeus* is restarted, it checks the existence of this file and loads it into memory if possible. In addition, a "prepare for shutdown" command can be manually given to the server to write its state to disk on demand.

This simple method suffices because the information retained in the server can be recomputed relatively quickly. Even if state is lost or not saved, the nature of the information is such that it can be recalculated by the client processes, so saving state should only serve as a backup in case that clients or their environments cannot be restarted.

4.5.3 Client-server mechanism

Zeus operates as a separate process and communicates with clients over the UNIX socket-based facility. It relies on a set of tools developed in the group called the *Socket Manager* (SM) and the *Byte Stream Manager* (BSM), which simplifies the implementation of inter-process communication [Aro92]. SM handles low-level connection I/O, supporting a call-back mechanism for connection initiation, termination and data transfer. BSM is an abstraction above SM, providing an RPC compiler and run-time library that supports syn-

chronous and asynchronous calls. On each end, application code uses BSM to communicate with the other process. The client-server architecture is illustrated in this diagram:

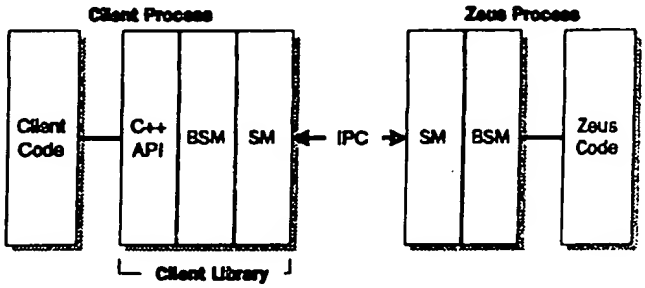

Figure 4.4: *Zeus*-client communication architecture.

The dtypes are transferred via BSM as binary streams, so additional mechanisms on both sides automatically convert dtypes to and from their binary representations.

4.6 Organization of Events

Information in the server is organized around a set of users and is stored on a per user basis. Each user in the group has a folder referenced by login name. For example, the information about user `lisa` is kept in a folder called `/user/lisa`. Embedded folders and records keep information about `lisa`; `lisa`'s list of new electronic mail messages is kept in `/user/lisa/email`. The organization of information is not established by the server but by clients producing the information.

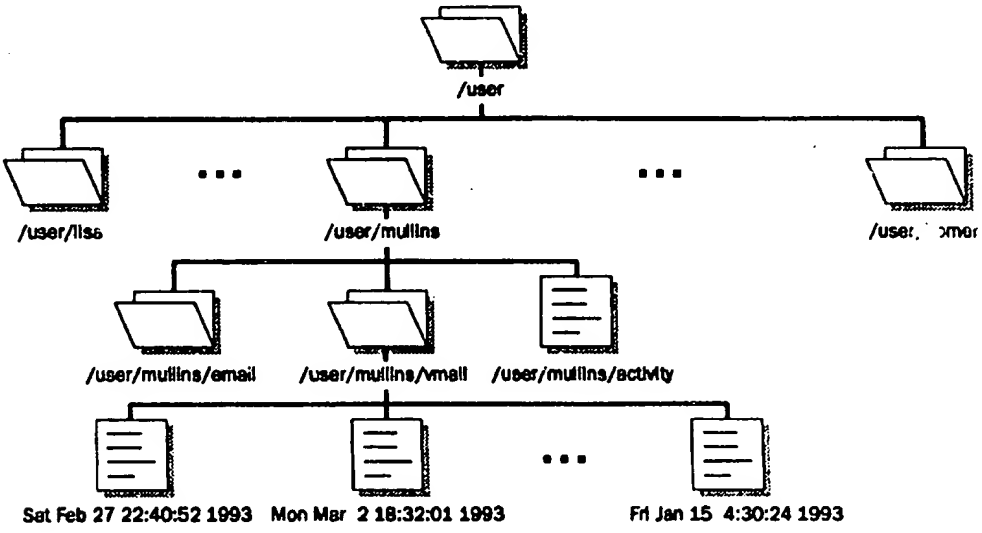

Figure 4.5: Present organization of the information in *Zeus*. Folder and record icons are shown.

At present, the following information is kept in *Zeus* and is maintained by a set of corresponding client processes.

4.6.1 E-mail and voice mail pollers

An `email` folder is used to keep a user's new messages. Messages are represented as environment dtypes with fields for the date it was received, the sender's mail address, the sender's name, recipient, subject and fields for accessing the actual contents of the message in the mail spool file. These fields consist of the beginning point of the message header, the beginning point of the body and the length of the body. Each message also contains a "rank" field representing the predicted interest of the message. (See "Collecting message features" on page 82 for a description of how the rank is computed.) In an more ideal setting, the actual messages should be stored in the server directly rather than requiring other processes to find them. Doing so would better fit our design criteria for providing better representations to user clients. Yet, we believed that the information in the server should be kept as compact as possible; such messages have a tendency to become rather large, especially if they contain multimedia attachments.

These mail records are constructed by an polling process which periodically checks a user's mail file to see if there are any new messages. Polling must be done on the file, but note that this process is the only one which polls the file. Messages are identified by the time and date during which they are received. Rather than rebuilding the entire list of message records in the server, only changes in the state of new messages are sent to the server. This has an important implication for daemons of new messages; they are only notified of changes in the mail file instead of getting a list of all new messages every time the update occurs.

Besides the telephone interface, a small program called *zbiff*, similar to the UNIX program *biff*, was written to use the server's notification system to inform the user of newly-arrived mail.

Records describing each user's voice mail messages are stored in a file on the desktop workstation [Sti91]. Similar to the above poller, another poller checks for newly-arrived messages and delivers records of them to *Zeus*. A record for each new message is stored in the server, with fields for message date, sender, filename of the recorded audio, message length, and the phone number of caller, if available.

Since the process of retrieving new voice messages is so similar to that of retrieving text messages, we made use of code inheritance. A generalized poller class is written with a specified API, and file access and server data construction methods are specified in subclasses to poll the appropriate file and create information for the server, respectively. The e-mail and

the voice mail pollers are thus implemented as a set of subclasses of the polling agent class.

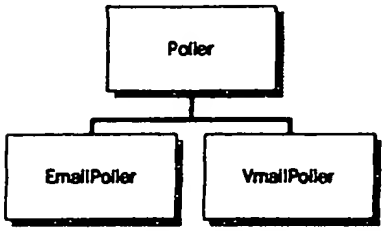

Figure 4.6: Poller class hierarchy.

The inheritance scheme makes it straightforward to implement other kinds of pollers which need to poll files to gather information.

4.6.2 Activity Information

A person's whereabouts and status are collected from active badges, lab workstations and telephones and analyzed by a set of processes, which report the final result as a record for each person.[1] Each user in the group has an activity record in the server. The record contains information from the group's Activity Server process, including the person's name, location (this could be a place or a remote machine), the last time the record was updated, the person's nearest phone, the host where the person is logged in to, the amount of activity on the host, and a time stamp of the last update.

The server can easily store other types of information, but due to time constraints only this information is stored currently.

1. Lorin Jurow carried out this work.

5. Speech Input

This chapter describes the design considerations and implementation of the speech input subsystem of Chatter. It discusses the design of Chatter's vocabulary, how parsing is then performed on this vocabulary to obtain a semantic representation to be used by the dialog system, and finally the need for having button controls for input even if fully conversational systems are possible.

5.1 Vocabulary Design Considerations

Since it is not yet possible for computers to interpret all human speech easily, the application designer must still decide what is possible and reasonable for an application to understand. The goal is to enable the user to speak naturally and freely into the system. However, some trade-off must be made for efficiency's sake. Chatter's vocabulary is created with the following motivations in mind:

- **Multiple ways of saying the same things.** One of the benefits of human-to-human speech communication is that great redundancy exists for giving the same orders. If an interface is to appear easy to pick up and use, it must mimic this redundancy by allowing users several ways of expressing the same thoughts. For example, to send a message to Chris, the user should be able to say any one of *send a message to Chris*, *send mail to Chris*, *send Chris a message*, *record a message for Chris*, and so on. The user should even be able to say *send a message*, from which point the interface engages him in a dialog to get the rest of the information. The vocabulary should have enough words and structure to support this multiplicity.

- **Conversational utterances.** Conversational speech is different than written speech. Spoken sentences tend to be more fragmented or incomplete. For instance, if the interface poses the question *send a message to whom?*, the user is not likely to say *send a message to Chris* but more likely to elide the answer by saying *to Chris* or even just *Chris*. The vocabulary should be designed to handle such fragments.

- **Faster speech input.** While speech is faster than typing, it is slower than buttons for certain functions. For example, imagine having to *say read the next message* several times to scan a list of messages. Not only is the utterance already long enough, having to say it more than once makes it tedious. The vocabulary should have some "shortcut" words for scanning information, like *next message* and *previous message* or even just *next* and *previous*.

The vocabulary is written for Chatter with these considerations in mind. It is given in "Recognition Grammar" on page 107. Although no user testing has yet been performed on the grammar or its vocabulary, it is adequate for prototyping purposes. Testing will be necessary if the application is put into actual use. One way to test the vocabulary is to draw up a questionnaire asking potential users how they would ask the system to perform certain Chatter tasks. The questions should also be framed in the form of a guided interaction, inviting fragmented responses from the user as well. Because a questionnaire cannot adequately capture all the states of the interaction, user testing must also be done on the running application. Transcripts can be recorded to determine the vocabulary's weak points.

5.2 Parsing Speech Input

The following section describes how speech input from the recognizer is parsed into a semantic representation. We assume that the user's spoken utterance is returned as a string from the recognizer, and the goal is to decompose it into a semantic representation useful for the dialog system to process. (The dialog system is described in the next chapter on page 54. Issues related to recognition are discussed in Chapter 8 on page 90.)

Parsing is performed with a simple, customized backtracking lexical analyzer in conjunction with *yacc*. Although English is not context-free, a limited but useful amount of spoken language, at least, can be captured by a context-free grammar. On the parser's side the grammar is specified with *yacc*, a UNIX tool for parsing Left Associative Left Recursive (with one token lookahead) or LALR(1) context-free grammars[1], and *lex* is frequently used with *yacc* to convert strings of characters into tokens that can be passed to *yacc* for processing. However, natural language is in general neither LALR nor context-free, and *lex* does not provide any backtracking capability on the token level. Usually, natural language requires backtracking to parse.[2] Therefore, some additional mechanisms are implemented to parse the spoken language input. A backtracking lexical analyzer is written to be used in conjunction with *yacc* to accomplish parsing.

One of the reasons a backtracking mechanism is needed is that English is filled with words that have multiple meanings. Many words fall into several syntactic categories. For instance, the word *reply* is both a verb and noun. The lexical analyzer allows words in the lexicon to be labeled with multiple categories. It is desirable not to limit Chatter's vocabulary simply on the basis of such category ambiguities.

To simplify parsing, the lexicon also accepts strings of words, called *word groups*, which

1. See [Aho88] for a description of LALR grammars. See [Mas90] for a tutorial on *lex* and *yacc*.
2. A chart parser is commonly used to parse English, but one was unavailable for this thesis.

the analyzer processes as single words. Word groups are groups of words that always occur together as a functional or conceptual unit, so it is also convenient to process them in this way. For example, *voice mail* is a word group that refers the a concept of voice mail, although *voice* and *mail* are separate words that have their own meanings. Other examples, are *e-mail message*, which means the same thing as *e-mail* (at least in Chatter's domain), and *phone number*, which means the same thing as *number*. Since *e-mail* and *e-mail message* refer to the same concept, they can be represented by the same parse tokens and reduce the need to model language syntax more closely.

Chatter's lexicon is given in a text file. Each entry presents a word or word group and its types and possibly subtypes separated by colons. For example, a partial listing of the lexicon is:

```
"email"                    OBJ:OBJ_EMAIL
"email message"            OBJ:OBJ_EMAIL
"message"                  OBJ:OBJ_MSG
"reply"                    VERB:ACT_REPLY, OBJ:OBJ_REPLY
"to"                       PREP
"the"                      DET:MOD_THE
```

Figure 5.1: Sample lexicon listing.

Here, the phrase *e-mail message* is a word group whose type is OBJ and whose subtype is OBJ_EMAIL. The type roughly corresponds to the syntactic category to which the word or word group belongs, and the subtype gives its "meaning." The word *reply* has two types: it is a verb with subtype ACT_REPLY and also an object with subtype OBJ_REPLY. These types correspond to actual token types for *yacc*.

When a string is received from the recognizer, the lexical analyzer uses the lexicon to make possible token labelings to words in the utterance. Because of the existence of overlapping words and word groups and multiple word types, there can be several possible labelings. Consider *reply to the e-mail message*, which, according to the above lexicon, can be labeled in different ways because *reply* has multiple types and *e-mail message* can be broken up into *e-mail* and *message* or taken as a whole. The possible labels for the utterance are:

reply	to	the	email	message
VERB:ACT_REPLY	PREP	DET:MOD_THE	OBJ:OBJ_EMAIL	
VERB:ACT_REPLY	PREP	DET:MOD_THE	OBJ:OBJ_EMAIL	OBJ:OBJ_MSG
OBJ:OBJ_REPLY	PREP	DET:MOD_THE	OBJ:OBJ_EMAIL	
OBJ:OBJ_REPLY	PREP	DET:MOD_THE	OBJ:OBJ_EMAIL	OBJ:OBJ_MSG

Figure 5.2: Sample labeling of parser.

The goal of the lexical analyzer is to find a labeling for which a non-backtracking parser like *yacc* can successfully parse the utterance. The analyzer uses a backtracking algorithm which iteratively enumerates over all assignments of word groups. Beginning with the first word or word group in the utterance, it finds the first possible labeling for the word group in the lexicon. It passes the candidate type and subtype label to *yacc* as a token. The algorithm also prefers to label the longest possible word group before fragmenting it. If *yacc* does not reject the token labeling passed to it, the analyzer continues to find labelings for successive words. Otherwise, it tries to relabel the word without changing the breakdown of the utterance. Finally, if simple relabeling does not work, then the algorithm recomputes the word groups in the utterance and attempts to relabel them again.

In the above example, no backtracking is actually necessary because the first labeling results in a successful *yacc* parse. While it is rejected because it is not defined in Chatter's grammar, the third labeling may also be considered valid in certain circumstances as a fragment to a question. For instance, it may be a valid reply to the question *send what to Chris?* If the grammar is extended to accept this utterance and the dialog's context prefers the third interpretation over the first, then *yacc* can be explicitly told to reject the first labeling, causing the lexical analyzer to backtrack until it finds the third, acceptable labeling.

The complexity of this lexical analysis algorithm is not good, although it is not at all severe for average cases. In the worst case, the complexity is the product of the number of possible word group labelings for the utterance times the number of permutations for assigning different categories to words. The longer the utterance, the higher the required processing time. Since the user's utterances will be relatively short, the worst case is not too bad. In many cases, there is only one possible labeling for the utterance.

5.2.1 Organizing information into a speech frame

The information is collected by the parser into a data structure that represents the meaning of the utterance. For this purpose, a data structure of type SpeechFrame is defined[3]. A SpeechFrame contains all the semantically useful information distilled from a speaker's utterance. It contains a main type followed by fields, which essentially serve as arguments to this main type. The type gives the kind of information in the spoken utterance—it can be an *action*, a *question*, an *acknowledgment*, or an *other* type. The motivation for choosing to use these types is simple: they seem to represent the range of utterances that a user may give in conversational systems, and by analyzing an utterance in this manner, it provides a convenient means of dividing the processing tasks of analyzing and acting on what the user

3. The SpeechFrame data structure is actually a C++ class. Besides containing the public variable members to be mentioned, it has methods for creating, destroying, initializing, copying and printing out of its contents.

has said. A function can be written to process all action commands, while another function can be written to process all questions, and so on. The *other* type provides a catch-all for other utterances types which do not logically fit in with other named categories.

The type of structure is always set, although the remaining fields only contain real values if it was given in the latest utterance. They assume an "unknown" value by default. A Speech-Frame contains these fields:

- **Type.** It gives the type of utterance: an action, a question, an acknowledgment or an *other* type. A sentence fragment is casted to the *other* type.
- **Acknowledgment value.** If the sentence uttered contains an acknowledgment of some form, either *yes*, *no*, or *ok*, then this acknowledgment value is placed into the field.
- **Action value.** This field gives the action if the user specified one. In Chatter, it may be to send a mail message, call someone, or read the next message.
- **Modifier value.** The modifier field records a modifier for the following object field, and it may contain either an indefinite reference such as *a* or *an*, a definite reference such as *the*, or values indicating *old*, *new*, *next* or *previous*. With the exception of the adjectives, the articles are more often than not confused by the recognizer so their use is unfortunately severely limited.
- **Object value.** The objects of Chatter are the objects of the application, a message, a name, an address and so on. In addition, a special value indicates the pronoun *it*.
- **Person value.** If a person was given in the utterance, the string of the person's name is given here.
- **Other value.** Like the *other* type, this field is intended as an appendage for information which does not fit into the other fields. Currently, it is being used to return a mail format when sending messages.
- **The utterance itself.** An exact copy of the recognized string is included in the structure in case the dialog system needs access to it. Currently, the only use of this field is to inform the user what the recognizer heard in case the utterance cannot be processed in any way in the context of the discourse.

For example, the above utterance *reply to the e-mail message* generates the frame, where the above fields are given in order:

```
[Action n/a ACT_REPLY MOD_THE OBJ_EMAIL n/a n/a
   "reply to the email message"]
```

The given SpeechFrame is rather simple because of the application domain, but it is meant to be reconstituted to contain richer information should an application domain require it.

5.3 Speech and TouchTone Controls

While speech is intended to be the primary interface medium, Chatter also enables the user to control the application with both speech and a TouchTone interface. The availability of speech input does not supplant the use of button controls, such as TouchTones on the telephone, in conversational systems, and this section addresses the reasons for needing both.

The Chatter user interface combines multiple complementary input and output modalities. Chatter can be operated using speech alone or buttons alone or a combination of the two. The command layout of *phoneshell* is retained in Chatter as it is independent of the issues in this thesis.

The implication for conversational systems is that the discourse management system is not only updated by speech input but also by button control. Pressing a button to initiate an action should make the appropriate updates to the discourse model. Fortunately, button input tends to be less sophisticated than the expressiveness of speech, so adding button input may involves more of a grafting process than a redesign.

6. Modeling Dialog

Perhaps the most important aspect of designing conversational interfaces is the creation of dialog models—how an interface manages to track what the user said in the past and presently fits them into the context of a discussion. This chapter describes a framework for constructing conversational interfaces. It reviews the well-known Grosz/Sidner discourse model for task-oriented dialog and presents a design for implementing the theory's key ideas. It outlines how this framework is applied to construct Chatter's dialog system, addressing issues of building discourse systems in the context of imperfect speech recognition systems.

6.1 Grosz/Sidner Discourse Model

The Grosz/Sidner theory was first proposed for task-oriented discourse involving participants accomplishing tasks.[1] According to this model, discourse can be analyzed into three interrelated components: a *linguistic structure*, an *intentional structure* and an *attentional state* [Gro86]. Together, they capture the external and internal structure of discourse:[2]

- **Linguistic structure.** The linguistic structure is the external decomposition of a linear sequence of discourse utterances into *discourse segments*, which serve as convenient elements for analysis. Like words in a single sentence forming constituent phrases, the utterances in a discourse are aggregated into segments, which fulfill certain functions for the overall discourse. These segments can be embedded within others to reflect discourse-related dependency relationships. The determination of segment boundaries is usually performed through the use of linguistic devices in the discourse, such as cue phrases (examples are *in any case, incidentally* and *therefore*), intonation and changes in tense. Boundaries can also be determined by observing changes in the intentional structure and/or the attentional state, changes in which signal changes in this linguistic structure.

- **Intentional structure.** Segments are associated with goals or intentions—the reason a particular segment is being discussed—and this structure is captured in the intentional structure. Discourse intentions are the internal goals or purposes that the current segment of discourse is being devoted to fulfilling by the segment's initiator. This structure

1. Another theory of conversation is proposed by Clark and Schaefer [Cla87] [Cla89], which describes the step-by-step construction of local structure, according to the status of mutual understanding. See [Cah92] for a computational treatment of these ideas. The Grosz/Sidner model was chosen because it is more comprehensive and simpler to implement.
2. The explanation in [Cah92a] provided a useful guide for the development of the ideas in this section.

roughly corresponds to the "logical" organization of sub-purposes and overriding purposes for having the discourse. The purpose for initiating the entire discourse in the first place is known as the *Discourse Purpose* (DP), and individual segments have associated *Discourse Segment Purposes* (DSPs), which are used to fulfill the overall DP. For example, certain things must be explained or introduced before other things can be said, as in the case of giving step-by-step instructions. Each step can be considered a segment with its own DSP, and the DP is the purpose of giving the instructions in the first place. These dependency relationships induce the embedding of segments in the linguistic structure. A segment begins when the DSP of a previous segment is satisfied and a new DSP is intended to be fulfilled. A segment terminates when its DSP is satisfied. Correspondingly, a discourse ends when its DP is satisfied.

While related, the intentional structure differs from the linguistic structure in that changes in the linguistic structure can signal changes in either the attentional state or the intentional state but not the other. The intentional structure only represents the DSPs of discourse, while the linguistic structure is a record of the entire discourse's history. The intentional structure also provides an abbreviated yet fully meaningful way for computational representations of discourse to be kept.

- **Attentional state.** The attentional state is a dynamic internal representation of the objects, properties and relations salient at each point in the discourse. This information represents the objects introduced into the discourse and known by the discourse participants. The state will be used to compute references to previously-introduced objects in case they are mentioned anaphorically.[3] Similar to the intentional structure, the attentional state also has different levels of embedding related to the objects introduced for the purposes of the discourse DSPs. Yet, the two are distinct because the intentional structure represents the full record of the intentions of the discourse, while the attentional state is intended primarily to represent the current attentional space.

6.1.1 Computational aspects of the theory

Computationally, the attentional state is the primary data structure which must be maintained during human-computer discourse. This state is represented by *focus spaces*, which store the objects of attentions of the currently active discourse segment. A focus space exists for each active segment of the discourse. Discourse segments can be nested, so focus spaces are placed on a *focus space stack*, which is pushed and popped with focus spaces during

3. Anaphora is a linguistic term essentially meaning some referring expression. It is a grammatical substitute used to refer to a preceding word or group of words. Well-known anaphoric expressions include *he*, *she*, *it*, *this*, *this one*, *that* and *that one*.

segment changes. The focus space stack is not used strictly as a stack because a process can reach through the top focus spaces to get information stored in lower spaces without removing the top spaces.

6.1.2 An Illustration of the theory

These concepts can be more clearly explained by an example. In the following simple dialog, the computer initiates a dialog with the user to read a message it has just received. The user is asked whether he would like to read Barry's message. The user asks about Barry's whereabouts, and upon getting an answer and a suggestion to call him, he decides to read the message instead:

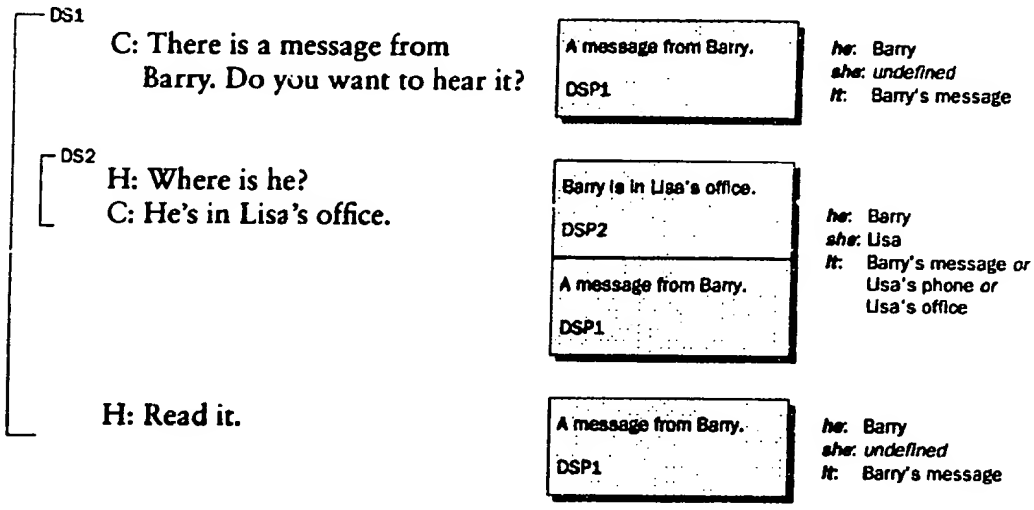

Figure 6.1: Changes in discourse structure. The dialog takes place between the computer (C) and human (H).

Though it is relatively simple, this dialog embodies all of the above concepts. The linguistic structure of the dialog is represented by the brackets in the figure. Discourse segment 2 is embedded in discourse segment 1 because DS1 is a continuation of the message-reading dialog, but it is a new segment because a new DSP has been established, namely the intention of locating Barry. To the right of the dialog is a display of the changes that occur in the focus space stack. After the computer's initiation, the stack contains one focus space, representing the objects in the discourse. DSP1 represents the intention of this segment—the computer seeks to play Barry's message. If the word *it* is used in an utterance, it would unambiguously refer to Barry's message at this point. If the pronoun *he* were resolved, it would refer to Barry. This knowledge can be used by both human and computer to shorten utterances and make interactions more efficient. In general, resolution of anaphora is a domain-dependent problem which requires domain-dependent knowledge.

With the question, the human introduces a new focus space, which is asserted on the top of the stack. The contents of the second focus space shows the state as it would be after the computer's response to the question. As noted, the DSP2 rer esents the intention to determine Barry's whereabouts. It seems that the *it* referent can be resolved to one of any number of objects, depending on the semantic requirements of the utterance in which it is used. If *read it* is spoken afterwards, then *it* clearly refers to the message, because it is the only object which can be read. Compare this utterance with *what's its number?* and *where is it?* The multiple use of a single pronoun makes the resolution of *it* and other anaphora more difficult because it requires domain knowledge of what operations can be performed on objects. However, this is not the standard theory of anaphoric interpretation.

Finally, the user ends DS2 by signalling that DSP2 is satisfied and returning to the main point of the dialog. DS2's focus space is popped off the stack, and its previous reference resolution in restored.

6.1.3 Modeling Interruptions

For spoken conversations, one of the most important aspect of any discourse theory is its account of interruptions—discontinuities in the progression of dialog. When tasks become even slightly compiex or involve several steps, the user will want to interrupt a lengthy task to perform another, whose completion may depend on the success of the original task.

The theory adopts three types of interruptions for performing tasks: true interruptions, flashbacks and digressions. An interruption induces a new segment, with corresponding changes in the focus space stack. True interruptions are discourse segments which have totally unrelated purposes or objects from the previous segment. An example is the following discourse: *are there any messages from Barry? Where's Chris? OK. Read them*, where the question *where's Chris?* is unrelated to the original discourse's purpose. In the computational domain, the most significant difference is that an impenetrable boundary must be placed below the focus space of the true interruption. This boundary prevents entities in the spaces below from being available to the spaces above. Flashbacks, the second type of interruption, are used for interrupting the flow of discussion because some purposes, propositions or objects needed to be brought into the discourse but have not been. This interruption type may be introduced by the linguistic cue *oh yeah, I forgot to tell you...* The third interruption type is the digression. Digressions commonly get introduced by *that reminds me, by the way,* or *speaking of ...* The digression segment makes reference to some entity that is salient in both the interruption and the interrupted segment and yet the discourse purpose is different, so in contrast to the way processing is done for true interruptions, only focus spaces associated with the interrupted DSPs are placed on an auxiliary

stack until they can be reintroduced into the conversation. The main stack is then used to track the digression's discourse.

We now apply the Grosz/Sidner theory to implementing a conversational model for Chatter.

6.2 A Framework for Organizing Conversation

Implementing a dialog system involves using domain-independent (the discourse theory) and domain-dependent code (the domain knowledge). The dialog system described in this thesis consists of two stages: it implements a general framework for building conversational systems, on top of which the dialog specific to Chatter is built. The goal is that the basic framework is general enough that it can be re-outfitted for other applications.

The implementation organizes discourse around a set of data structures representing segments. Since the three components of discourse interrelate, it is convenient to construct data structures representing segments of discourse and associated focus spaces. The basic approach we take is to divide the interaction into a set of *dialog segments*, each of which is represented by a data structure that maintains state of the segment's dialog. Each dialog segment roughly corresponds to a major task of the application. For instance, a segment exists for reading messages, another exists for sending and replying to messages, and so on. The segments also have computational capabilities for resolving pronouns, generating user feedback and subsetting the vocabulary for the recognizer.

The segment data structures are implemented as a set of classes in an object-oriented language (C++). These data structures are formed from a root DialogSegment class, which implements the discourse theory. The DialogSegment class provides a skeleton on which to build application-specific dialog segments. Instances of DialogSegment subclasses are placed on a stack owned by a Conversation object, which manages the entire conversation. The basic execution of a conversational application then consists of creating a Conversation object and initializing its stack with some default dialog segment, from which user interaction causes other segments to be pushed and popped from the stack dynamically.

In this framework, we will call the entity known as the focus space stack the *dialog stack* to signify that the theoretic model and implementation are not exactly the same. Similarly, the implemented version of a focus space will be called a *dialog segment*. Dialog segments are slightly different from the theoretic discourse segments because they combine both linguistic and intentional structures into the same structure. The state of the dialog stack represents both the linguistic nesting (for purposes of anaphoric resolution, say) of the dialog as well as its intentional state (goals of the current tasks). A *dialog segment*, *segment*, or Dia-

logSegment will be used interchangeably to refer to subclasses of DialogSegment. Below, we present the algorithms used in this framework in general terms and how they are used in Chatter.

6.2.1 Choosing appropriate segments

Conversing with an interface consists of many smaller conversations, and it is often necessary to divide the conversation into one or more segments of discourse. Choosing the appropriate breakdown of segments is not well defined. One rule of thumb is to create segments in such a way that each segment collects a major task of the application. A task can be defined as either:

- A function having informational requirements, or better, a command with associated arguments. For example, the task of sending a message requires some information before it can be executed: a recipient, the type of message (e-mail or voice mail), the actual text or voice message to be sent[3], the mail format if it is to be a multimedia e-mail message. These pieces of information must be known before the task can be executed.
- A dialog centered around some object or functionality. For example, the user may engage in a dialog with the interface to find information in his rolodex, consisting of queries and answers to the database for information. Another example is a calendar. In this case, the dialog may consist of finding out about or scheduling appointments.

Some tasks may be quite large and complex and require sub-tasks, so it may at times be desirable to define new segments which manage sub-tasks. If the interaction in the sub-task is likely to be complex, creating a new segment makes the dialog modeling less complex and more easily extensible.

These ideas inspired the segments of dialog created for Chatter. Eight segments are created, and they encompass the Chatter's interaction with the user. The purposes of each segment are explained below:

- **BaseSegment.** Always resides as the bottom of the dialog stack to serve as the default segment. This segment does little else than ask the user what he would like to do.
- **CallSegment.** Manages the task of calling a person. Chatter allows a call to be placed from within the application. If the user wants to cancel out of the call, he may do so by saying *Chatter, hang up*. The user is then returned to the interface where he can per-

3. The message may be represented as a pointer to a file containing text or audio.

form other tasks.[4]

- **ComposeSegment.** Allows the user to compose an e-mail or voice mail message. This segment guides the interaction for sending a message in any situation, such as sending a new message, replying to a message, or forwarding a message. Because users have different media from which to receive messages (say, voice, text or various mail addresses), the segment also contains knowledge about how to best deliver messages to users based on the information in the user's rolodex as well as other databases. It queries the user about mail delivery methods it cannot find.

- **ExitSegment.** Becomes active at the end to terminate the conversation with the user. Currently, it does nothing more than understand *good-bye* as the command for termination.

- **GreetSegment.** Initiates a Chatter session with the user. This is the initial segment which helps establish the identity of the user; it is the top segment of the dialog stack when a Chatter session begins. The segment asks for the user's name and then asks for him to speak a secret password or type in a security code.

- **HoldSegment.** Handles the dialog for putting Chatter on hold. It may be desirable to suspend a dialog with Chatter from time to time because the user is being interrupted by external events, such as conversations with other people, loud background noise, or the need to turn attention to another task. This segment is initiated by the user speaking *stop listening* and completes when the user says *pay attention.*[5]

- **PersonSegment.** Responds to user's questions about a person's information, such as his e-mail address, work and home addresses and phone numbers. It also answers questions about a person's whereabouts. To answer these questions, this segment uses the information stored in the user's rolodex as well as a set of active badges and UNIX *finger* sensors.

- **ReadSegment.** Handles the reading of electronic and voice mail messages. It allows the user to scan among a list of messages, read and save messages, and to query about various aspects of a message, including its subject, who it is from and the time it was

4. An interesting idea is to have Chatter remain on-line during the phone call to serve as an assistant who is listening in on the conversation. This assistant may be called on from time to time to provide various kinds of information for both listeners (e.g. *Chatter, what's Lisa's phone number?*) or for the user to perform other tasks in the presence of the other listener. Since the conversational dynamics of a three-way conversation over the phone are different than a one-to-one exchange, we may be able to exploit some pre-established conventions to avoid insertion errors with the recognizer. For example, we could require that the user to say *Chatter* before initiating a dialog with the agent interface. Other methods need to be developed to detect the end of a dialog with Chatter.

5. Time permitting, this segment should also allow users to ask questions like *where was I?*

received. This segment also incorporates some intelligence to suggest "interesting" messages for the user to read first. See the next chapter for a description of the learning algorithms that makes this suggestion possible.

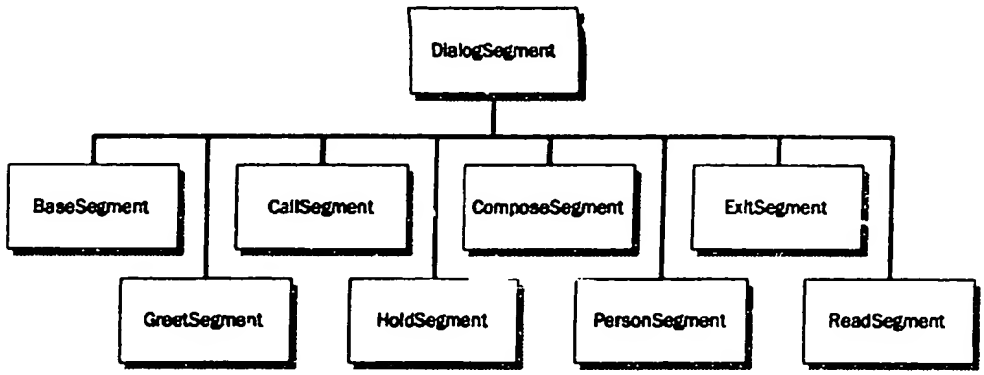

Figure 6.2: **DialogSegment** hierarchy for Chatter. The lower eight segments are specializations of **DialogSegment** which implement the discourse for Chatter.

6.2.2 Responding to speech frames

This section outlines the way in which the dialog framework processes user input. It assumes that linguistic processing has already been performed on the recognized input and the information is provided in some semantic representation. In the last chapter, we explained how Chatter converts recognized speech into semantic **SpeechFrame** data structures (see "Organizing information into a speech frame" on page 51), which the dialog system then manipulates:

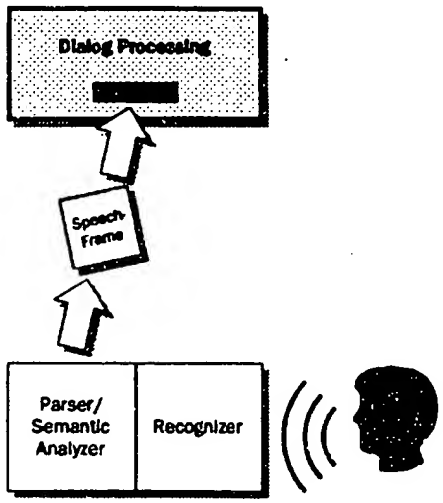

Figure 6.3: Sketch of the speech system.

Processing a SpeechFrame is the responsibility of the application's DialogSegments, the classes implementing the application's specific dialog models. These segments are a convenient place for such processing because they already implement the necessary domain-specific knowledge for the dialog. Each segment has individual methods for receiving the various SpeechFrame types.

The dialog stack is initially empty, and segments are activated and pushed on the stack as they are needed to process SpeechFrames. Upon receiving a frame, the dialog system first looks on the stack to find an appropriate segment that can process it. The algorithm for finding the appropriate segment to process a frame is as follows.

1. Send the SpeechFrame to the top segment on the stack and see whether it can process the frame. If so, the segment performs the appropriate execution and returns a value indicating that it has processed the frame and the algorithm is finished.
2. If the top segment cannot process the frame, then try the frame with the next lower segment on the stack. Continue to do so until the a segment has been found which can process the frame.
3. If no segment on the stack can process the frame, then see whether one of the inactive segments can process the frame. Each segment class implements a function which determines whether the segment can respond to a given SpeechFrame. The Conversation object finds the first segment class which can respond to the frame, instantiates it and pushes it onto the top of the stack.

As a simple execution example, consider the following scenario where the user speaks *read my messages*, hears the first message and says *send him a reply.* Initially, the dialog stack contains only BaseSegment:

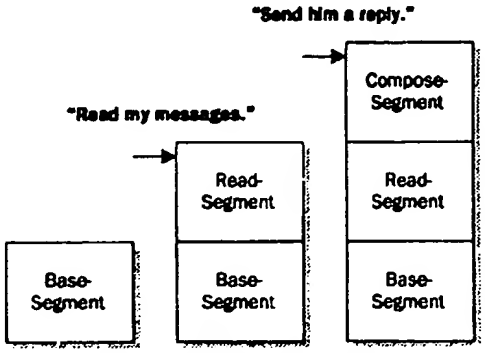

Figure 6.4: Example of dialog stack operations.

After receiving a SpeechFrame for the first utterance, the system realizes that none of the segments on the stack can process it. It finds that the ReadSegment class can process the

frame, so it instantiates the new segment and pushes it onto the stack. At the second utterance, the system realizes that neither **ReadSegment** nor **BaseSegment** can respond to the second frame, so it finds that the **ComposeSegment** class can respond, instantiates one, and pushes it onto the stack to handle the frame.

6.2.3 Resolving pronoun references

As in a focus space, each segment also records objects, relations and properties that can be used in resolving anaphora. The root **DialogSegment** class accommodates this storage by having functions for resolving pronouns. As mentioned previously, pronoun resolution requires domain-dependent knowledge, so it is only possible to provide a domain-specific algorithm. Currently, the system is unsophisticated[6]; it defines only references for *it*, *him* (or *his*) and *her*. The pronouns are tokenized as `REF_IT`, `REF_HIM`, `REF_HER`, respectively. `REF_IT` also has sub-types, such as `REF_MSG` or `REF_PHONE`, for correctly resolving multiple possible *it* references.

Each segment maintains a database of possible references, which it is responsible for keeping up-to-date as the state of the its discourse changes. An asserted token has an associated application-defined record giving information about the referent. For instance, as a user moves from message to message, a **ReadSegment**'s references for `REF_IT` (with sub-type `REF_MSG`), `REF_HIM` or `REF_HER` changes. `REF_IT` dereferences to a record of the currently-read message. This record contains the message's file path, message type and sender's address.

When a segment wants to "borrow" a reference from another, such as a **ComposeSegment** being invoked by *forward it to Gayle* from a **ReadSegment**, the following simple algorithm is used. It begins at the top of the dialog stack and searches segment-by-segment until it can find a reference matching the description. If none can be found, a null reference is returned.

6.2.4 Terminating a segment

Determining when a segment should complete and be popped off the dialog stack depends on whether the purposes of the segment have been fulfilled. With task-oriented discourse, the answer depends on whether the task at hand has completed. There are several issues to consider: some tasks have "logical" end points, while for others, such points may be less clear. For those tasks having natural breakpoints, the interface may terminate a segment

6. With the possible exception of the **HoldSegment**, the modeling of true interruptions, where impenetrable boundaries are placed below the interrupting segment so that incorrect resolution does not occur, is not implemented due to time constraints.

when such points arise. For the ambiguous case, deciding when to terminate is an issue of arbitration between the user and agent. Either the user or agent can terminate a segment, so the problem reduces to whether the user or interface is *driving* the dialog. Fortunately, some tasks are amenable to being more user-driven while others are more application-driven. For instance, it is fair to construct the **ReadSegment** so that it does not terminate until after all messages have been read (or it has been told not to present more messages). The interface is driving the completion of this segment. Likewise, a **ComposeSegment** does not terminate until a message is delivered. In the case of a **PersonSegment**, the segment completes as soon as it receives a **SpeechFrame** for which it cannot process. It assumes the user has gone on to another segment and terminates. Consequently, the user is driving its completion by asking the questions. When the arbitration is more arbitrary, some learning mechanism can be used to discover the user's habits and discover plausible stopping points. This is an area of future work.

In the implementation of **DialogSegments**, a segment decides when it should self-terminate. The decision to terminate is usually a result of processing the latest speech frame or a timer expiring. A **DialogSegment** issues a **terminate** message to itself, and the dialog mechanism will remove and free it from the dialog stack at the end of the event cycle.

Even for segments with natural termination points, determining when to terminate is not as straightforward as discussed above. Often, leaving a segment active on the stack after the task has completed is necessary because the user's next request may force an otherwise terminable segment to remain active. For example, a **ComposeSegment** invoked for replying to a message may be considered terminable when its message is delivered. Yet, the user may want to continue the topic by saying something like *send the reply to Atty*, forcing the segment to remain active because *the reply* needs to be resolved. (In general, the user may actually want to continue or refer to focus that is long past in the discourse. In this case, a more complete history of the discourse needs to be memorized and algorithms for calculating references. This may be the topic of future work.)

One simple but perhaps inelegant method for solving this problem is to leave the completed segment on the stack but to set it to some *terminable* state. If the next speech frame does not require the use of this segment, it is then terminated and actually removed from the stack. ComposeSegment uses this method for determining its termination condition.

6.2.5 Asking questions

One aim of having a conversational model is to make the system more active in its interaction, and asking the user questions is one method for achieving this. Chatter uses questions

mainly for two purposes:

- **Eliciting missing information.** Tasks requiring a number of arguments may not be given by the user all at once, so any missing information is solicited using questions.
- **Making suggestions.** The system makes a suggestion based on its knowledge base and asks for user confirmation before executing a task or accepting the suggestion. Here, the reply may be a simple *yes* or *OK*, but it may also be a correction to the suggestion, like *no, Barry.* We deal with the topic more deeply in "Repairing Errors" on page 69.

Computationally, posing a question always occurs inside a segment because the question either introduces a new focus space or is part of the current one. Asking a question invites an answer, implying the expectation of an answer. Maintaining expectations can be represented as the setting of state inside a segment. Within Chatter segments, this state is represented most conveniently by a state variable. Another method is to represent state by the execution state, such as a modal loop. The reason for using a variable over a modal loop is that the user may not always answer the question immediately. Therefore, the interface should not be modal, expecting an answer before something else can occur. The state serves as additional context in interpreting user input. For example, a response of *OK* must be interpreted as an answer to some particular question. A segment must enumerate its questioning states so that responses can be interpreted.

Here is a dialog with Chatter, which illustrates many of these concepts:

H: Send a message.
C: Send it to whom?
H: To Barry.
C: Send Barry voice mail?
H: OK, voice mail.
C: Record your message now.
H: *records a voice message.*
C: *plays back the message and asks,* Go ahead and deliver it?
H: OK.
C: Sent.

Figure 6.5: Example dialog with questions.

In the above example, the interface asks questions to fill in the missing information needed to deliver the message. When it asks whether the user would like to send Barry voice mail, the system is employing domain knowledge to make a suggestion. By looking in a voice mail database, it determines that Barry is a local voice mail user and assumes that he prefers receiving voice messages instead of decoding an audio e-mail message.

6.2.6 Handling Interruptions

When handling questions and tasks, segments may possibly be suspended so that a sub-task can be performed. Interruptions are disconnected flows of interaction. In this dialog framework, they are detected when a segment responding to a SpeechFrame is not at the top of the stack. In this case, all segments above it are considered to be interrupted. When this occurs, an auxiliary stack is created, and these segments are temporarily stored on the auxiliary stack. When the interrupted segments are reintroduced, they are returned to the main stack. Continuing with the example in "Example of dialog stack operations." on page 62, suppose that while the user is sending a reply, he asks *what's the next message?* The ComposeSegment he was using for the reply is interrupted because the lower ReadSegment processes the command, so the ComposeSegment gets placed on an auxiliary stack. When he reintroduces the topic by saying *finish that reply*, the ComposeSegment is replaced on the main stack and the auxiliary stack is deleted.

At the present time, it is not known whether more than one auxiliary stack is needed, since it seems reasonable that an interruption should be interruptible. The system currently allows multiple interruptions by maintaining a series of auxiliary stacks. However, in actual practice people may only rarely interrupt an interruption.

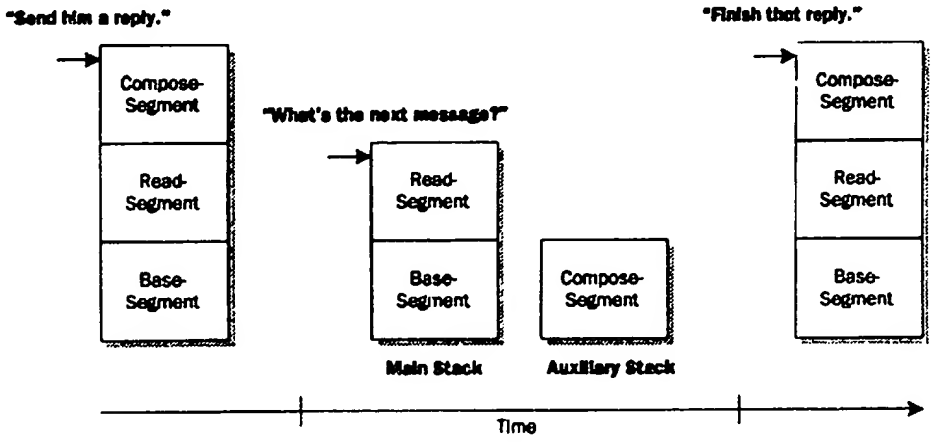

Figure 6.6: Interruption on the dialog stack.

6.2.7 Reintroducing Interrupted segments

The ability to interrupt segments creates the need for mechanisms to reintroduce interrupted or unfinished segments. Reintroducing a topic can occur in two ways. First, the user may want to return to a segment that was previously interrupted. The segment to which he is interested in returning is below the top of the stack or in one of the auxiliary stacks. For instance, the user is in the midst of replying to a message using a ComposeSegment. He

then asks for the sender's phone number, invoking a PersonSegment. Then he says, *ok, send the reply to him.* The phrase *the reply* effectively ends the PersonSegment segment and reintroduces the previous ComposeSegment by referring to the definite reply. In the second case, the application may want to reintroduce a segment. A segment has completed and is popped off the discourse stack, leaving an old active segment on top of the stack. One example of this situation is one where the user has just finished sending a message with a ComposeSegment. He was reading mail previously with a ReadSegment, so the interface may ask him whether he would like to continue reading messages. This question serves as the reintroduction of the unfinished task of reading mail by the interface.

Implementation-wise, allowing for segment reintroduction means two things: first, the algorithm in 6.2.2 is modified so that any interrupted segment is allowed the chance to process a speech frame before the dialog system instantiates a new segment. Second, the vocabulary for reintroducing a segment must remain active in the recognizer. This vocabulary set may be more restrictive than the one used when the segment is on top of the main stack. Some segments have explicit reintroductions while others have implicit ones. For example, an interrupted ComposeSegment can be reintroduced by *finish that message* or *finish that reply.* For the ReadSegment, reintroducing an interrupted segment is implicit: the same command for reading another message while in the mail reading context can also be used to reintroduce the task, such as *what's the next message?*

6.2.8 Subsetting vocabulary

Subsetting the space of words is a necessary step to make recognizers function more accurately. The possible interaction of any speech application is defined by its vocabulary, yet not all words of the vocabulary may be appropriate at all times in the interaction. Pruning away inappropriate vocabulary makes the recognizer more accurate and faster, since fewer word choices need to be tested.[7] A simple instance is the case where the utterance *yes* does not make sense at the start of a conversation.

A vocabulary set usually corresponds to its task, so it is convenient to organize subsetting functionality around dialog segments. Note that a segment may also have multiple vocabulary sets that can be selectively activated, depending on its state. As the state of the dialog changes, the state of the active vocabulary correspondingly changes. At any time, the active vocabulary is the union of all vocabularies of the following segments:

- **The currently active segment.** Vocabulary for carrying on any dialog relevant to this

7. This section assumes a particular style of recognizer which uses a constraint grammar for recognition. Clearly, one of the biggest drawbacks is that speech out of the vocabulary is either not recognized or incorrectly done so, so care must be exercised to design the vocabulary and how it is subsetted.

segment should be activated.

- **Other segments on the main stack and interrupted segments on auxiliary stacks.** The vocabulary for reintroducing these segments should be active.
- **Uninitiated segments.** Similar to the previous, the vocabulary for initiating one of these segments should also be active.

The dialog framework facilitates vocabulary subsetting in a simple manner. Each segment only needs to determine what vocabulary set it needs to have activated. The rest of the system computes the correct union based on the segments' location on the dialog stack as described above. See "Using start hypotheses" on page 92 for more details on how subsetting is accomplished with Chatter's recognizer.

6.2.9 Speech generation

The dialog system uses a straightforward scheme to generate responses and queries based on a template-matching method. Each DialogSegment maintains a set of templates, which are used to generate actual responses. A template is a complete utterance with temporary placeholders called *keys* which can be filled in with information at generation time. A key is any word preceded by a `%` symbol. These keys correspond to the information to be echoed. For example, a template for responding to a user who has asked to send mail may be:

```
send a %msg-type message to %who? ("%msg-type" "%who" )
```

where `%msg-type` is a key for the type of message to be sent, either "`voice`" or "`email`" for instance, and `%who` for the recipient. Also associated with each template is a set of keys, any of which may or may not be mentioned in the template's string. This set of keys specifies the exact set which needs to be asserted for the template to be used. In the above example, the required set is given in parentheses. Not every key is mentioned in the string because not everything needs to be literally echoed to the user. Each segment contains a database of such templates.

In addition, segments maintain a database of collected keys with associated values that contain the information provided by the user in the last utterance. If templates are provided, generating appropriate speech feedback becomes an relatively task. During semantic analysis, keys containing information are collected into the database. The order in which keys are asserted into the database does not matter. Once analysis is done and it is time to generate a response, a function is invoked to match the key database with the template set. The first template whose associated keys match exactly those asserted in the database is chosen as the template to be used for generation. All keys in the string are replaced by the key's

value and then spoken. The database is cleared again in preparations for the user's next utterance.

6.2.10 Using keys for directing dialog

Not only are keys a useful means of echoing information, they can also be used as a means of directing conversation—such as reporting the result of a task or asking for additional information. Since every template has a required-key set and new key types can be arbitrarily created, they may be used as an simple way to relay execution state to the user. For example, the above template can be extended to a success and failure form:

```
sent a %msg-type message to %who. ("%msg-type" "%who" "%ok")
unable to send a %msg-type message to %who.
                                  ("%msg-type" "%who" "%fail")
```

Then, completion of the composing task would also include the assertion of either `%ok` or `%fail` into the database, and the success of the operation can also be given to the user in the same utterance.

This template-based approach has an important advantage: it separates the information to be spoken from its syntactic construction from a per-utterance standpoint. The templates provide the syntactic structure for building responses while the key database maintains the information the system needs to communicate. The tasks of information construction and syntactic construction are then separated and can be performed more easily on their own.

Clearly, if many different combinations of information exist in the dialog and causes variations in sentence structure to become large, then the number of templates needed also becomes excessively large. In the case of this scaling problem, however, the method can be extended so that templates can also serve as subparts for other templates. For the given class of conversational speech systems, this basic approach appears to be very usable.

6.3 Repairing Errors

Detecting and repairing error is perhaps one of the most difficult problems to solve in interactive systems. A system which remembers the history can become quite confused if it remembers false information from sentences that were incorrectly recognized. The general problem is that user or computer may develop different assumptions and then follow a line of interaction, only later discovering that some previous assumption was incorrect. How do the user and interface untangle its misunderstandings and consequences so that both once again agree? To attempt to answer this question, it is necessary to examine the types of

errors which can be generated.

6.3.1 Types of errors

Assuming the interface makes no incorrect deductions[8], we can point to these types of error, most of which are made at the recognition stage.

- **Insertion error.** The user did not say something, yet the recognizer reports a result. This error occurs when the user intends to speak to someone else and his voice is detected by the recognizer or there is excessive background noise.
- **Substitution error.** The user says one thing but the recognizer reports another. With continuous speech recognizers, this type of error occurs either on a single word (like a name) or the entire utterance,
- **Rejection error.** A user's utterance is not recognized because of background noise or unknown words.
- **The user misunderstood or changes his mind.** The user misheard some piece of information generated by the interface or wants to change his mind about something in midstream. Perhaps he started a task not knowing all the correct information.

Because the recognizer uses a constraint grammar, it should not be possible to have parse or semantic error in Chatter. A constraint grammar defines the possible vocabulary space, so in theory, the application ought to be able to account for everything that can be spoken.

6.3.2 Commands for repair

Chatter deals simply with errors and provides only minimal mechanisms for error repair. First, because it is impossible to tell when any utterance is actually correctly recognized, any information is always echoed back to the user. Hopefully, this echoing providing a chance for the user to detect an error if one has occurred. For insertion errors, we implement *stop listening* and *pay attention* commands, described above briefly. Upon hearing *stop listening*, the interface suspends itself until a *pay attention* is heard. These commands allow the user to effectively turn off the interface to stop extraneous insertion errors.

Rejection errors unfortunately cannot be handled satisfactorily because the system cannot detect any input and assumes that the user has placed the interaction on hold.

8. Chatter's domain is simple enough that this assumption can be considered to hold, but it does not in many domains.

Substitution errors are the most challenging to repair. Consider the dialog:

H: Send e-mail to Barry.
Computer heard send e-mail to Eric.
C: Record your e-mail message for Eric.
H: Send e-mail to BARRY. *(with emphasis)*
...

Figure 6.7: Dialog with misrecognition.

In the above example, *Barry* is misrecognized for *Eric*, so the user repeats the entire original utterance hoping to clarify his request. Assuming the recognizer correctly recognizes at this point, the problem is that the system does not know whether the user has interrupted the task with a new one, or he is trying to make a repair. To avoid the problem of resolving the discrepancy, the following convention for repairing information has been adopted. Whenever a piece of information is misrecognized, the user can repair the incorrect information but preceding a correction by *no*. In the above example, the mistake can be corrected by *no, Barry*. Of course, this correction is also verified by the response *send it to Barry instead?*

More serious errors or misunderstandings can be cancelled by a *scratch that* command, which allows the system to completely forget all information that was newly introduced in the most recent utterance. This roughly corresponds to an "undo." This command is distinct from a *cancel* or *never mind* command, which has the effect of aborting the active segment.

Sometimes, a user may simply want to find out where he is in the interaction, so *help?* or *where am I?* commands gives the user a description of the state of the interaction.

7. Capturing User Preferences

As programs and the information they access become increasing complex, interfaces must become more active in alerting the user to the appropriate capabilities of the interface at the appropriate times during an interaction. However, implementing a strong user model has its drawbacks: it takes lots of time, and users are different from one another. This chapter describes the machine learning approach taken to user modeling in Chatter. It outlines the memory-based reasoning (MBR) algorithms, the main method used to capture user interests and preferences. The chapter then discusses how this algorithm is integrated into the conversational framework presented in the previous chapter. Learning is used to infer user interests about messages and support interaction in the context of dialog segments.

7.1 Background

Three approaches for building interface agents can be distinguished. We outline these approaches and related work from the oldest to newest. Most of the systems which perform inferencing on user actions have been termed *keyhole recognition*, referring to the computer attempting to look through a tiny channel to figure out what the user is doing.

7.1.1 User-configured rules

Perhaps the most basic approach is to have the user specify rules for how he would like the system to behave under given circumstances. There are several systems which demonstrate this method. Systems like Oval explore this method in great detail [Mal87] (see "Oval" on page 32 for a description). Another rule-based system is the widely-used public-domain program *Procmail*, which can be used to filter and sort incoming Internet e-mail into user mailboxes [Van93]. Regular expressions can be specified in a user configuration file to match message header files or some content in a message's body. These rules are invoked as soon as new mail arrives. The rules also have associated actions, including delivery to one of several folders or some arbitrary command execution. The trade-offs of this approach are discussed in the section describing Oval.

7.1.2 Plan recognition

Another approach, described briefly in Chapter 3, seeks to model the domain more strongly by directly encoding the intentions of users and inferring why they are performing certain tasks. This approach is known as the knowledge-based approach, which consists of endow-

ing a system with a domain-specific background knowledge about its application and user, often called the *domain model* and *user model*, respectively. This approach is adopted by the majority of people working on intelligent user interfaces. At execution time, the interface agent uses its knowledge about the domain to recognize a user's plans and find opportunities for contributing to them. The UCEgo system is one system which uses this technique for making inferences about what UNIX novices are trying to do (see "UCEgo" on page 33).

7.1.3 Knowledge-Based Agents

The third approach actually uses machine learning techniques to construct the user model. Two examples of this method, both by Maes and Kozierok [Mae93], are outlined in "Learning interface agents" on page 34. While there are several approaches to machine learning, the authors use memory-based reasoning and reinforcement learning algorithms to make inferences about new situations based on a memory of past, similar situations. Chatter relies only on the memory-based reasoning aspect.

7.2 Motivations

In order to work for machine learning to work effectively from an interface design standpoint, Maes notes that several conditions need to hold in the application domain:

- The use of the application has to involve a lot of repetitive behavior. If this condition is not met, a learning agent will not be able to learn anything because there are no regularities in the actions to learn from.
- This repetitive behavior is different for different users. If the latter condition is not necessarily met, i.e. the repetitive behavior demonstrated by different users is the same, a knowledge-based approach might prove to yield better results than a learning approach.

In general, these conditions hold for the intended use of Chatter. First, the system is designed for a group of regular subscribers, instead of casual users who only occasionally use the system. Second, different users will have different usage habits. For example, some users may only want to scan through their list of messages. Other users may want to read all their messages. Still, others may choose to read messages depending on who sent it. Such differences among users apply not only to the actions they take but also to the information in which they are interested.

Knowledge acquisition, or the process of identifying and formalizing rules, remains an art.

Knowledge engineers must collaborate with expert informants in the application domain, training them to think about their decisions and actions in terms that can be turned into rules and knowledge bases. This process often takes a long time, and in the end, the performance of such systems has often been disappointing.

Maes and Kozierok argue that machine learning also offers several advantages over hand-crafted rules. First, it requires less work from the end user and application developer. Second, the agent is adaptable over time to changing user preferences by automatically customizing behavior to suit individual user preferences and habits. Third, a particular advantage of MBR is that it allows the agent to provide explanations for its reasoning by justifying its recommendations based on past similar situations. The task domain is simple enough such that a more sophisticated method such as plan recognition need not be used to model the user. Yet, one of the disadvantages of machine learning is that training is required. The agent interface cannot recommend any courses of action that may be helpful unless the user has already executed them before. From a usage standpoint, the interface's naïveté does not bode well for a user unfamiliar with the capabilities of the program.[1]

7.3 Memory-Based Reasoning

In this section, we detail the memory-based reasoning approach, hoping to instill readers with strengths and weaknesses of this approach. It is hoped the reader will realize that MBR is only one of several methods that can be used to construct interface agents.

7.3.1 Using MBR

MBR, as described by Stanfill and Waltz [Sta86], works by representing situations as vectors of features. These features are computed and stored in a database of situations over time. Part of the recorded situations are the actions that the user took in them, so these actions are the source for making suggestions to the user.

For example, in the case of suggestions about e-mail, the following seven features are used to represent the situation: the subject string of the message; a boolean indicating whether it is reply to a previous message; the sender; all the recipients; a boolean indicating whether the mail is directly addressed to the user; the mail domain of the sender; and a quantized message length from "very short" to "very long."[2] The action is represented by six values indicating the users interest in the message. These values range from the user wanting to

1. This problem can be helped by engaging learned preferences from similar communities of users and assuming that some preferences of the new user will be the same as more established users. See [Orw93] for an application of this idea.
2. Another field may be a boolean indicating whether the message body contains the recipients name.

delete the message to the user being very interested in reading the message. Details on how these features are computed and other recorded situations are described later in "Integrating Learning in Chatter" on page 81.

7.3.2 Comparison with other learning approaches

MBR's key idea is to reason based on similar situations. The goal of this method is to make decisions by looking for patterns in data. There are several other related approaches with this flavor. It has been studied extensively in the context of *similarity-based learning* [Mic83], which uses observed patterns to create classification rules. MBR eliminates the rules, solving the problem by direct reference to memory. The two approaches differ in their use of the memory store: rule induction operates by inferring rules that reflect regularities in the data, while MBR works directly off the database. MBR also degrades gracefully when it cannot come up with a definitive answer to a problem: using a confidence measure, it may respond that no answer is possible, give one or more plausible answers, or ask for more information.

The other area of similarity-based induction is the *explanation-based learning* paradigm [DeJ86]. This is an incremental, knowledge-based approach, in which a system learns by explaining unanticipated events. The goal is "one-shot learning," in which learning can proceed from a single example. Explanation-based learning presupposes strong domain models, which are often implemented through rules or through data structures traversed by a global deductive procedure.

Work on *case-based reasoning* is also relevant [Sch82], as it combines memory with explanation-based learning to allow a program to learn by remembering and analyzing its mistakes. In this approach, when faced with a difficult task, a problem solver will search memory for a previous case where an analogous task arose and then try to adapt that solution to its current needs. Like EBL, it also requires a strong domain model. [Sta86] gives an excellent motivations for choosing MBR over more traditional AI methods and enlightened comparisons of MBR with other AI and cognitive approaches to learning.

7.3.3 Description of algorithms

The matching nature of memory-based reasoning can be reduced to the problem of finding the distance between two given situations in a situation space. Comparison between situations is performed by counting combinations of features, using these counts to produce a metric, using the metric to find the dissimilarity between the current problem and every item in memory, and retrieving the best matches. If a situation similar to the one at hand

can be found, it is reasonable to expect that some unknowns of the current one will be the same or similar to the past situation. If we consider situations to be points in this space, then their distance is 0 if they are the same. If they are not, then the distance between two situations should indicate approximately the similarity of the two according to some intuitive standard. Therefore, we require a *distance metric* that takes two situations and computes the distance between them.

Briefly, a distance metric Δ for points a, b, c is defined as:

$$\Delta(a,b) \geq 0$$
$$\Delta(a,b) = \Delta(b,a)$$
$$\Delta(a,a) = 0$$
$$\Delta(a,b) + \Delta(b,c) \geq \Delta(a,c)$$

The distance between any two points is greater than or equal to zero. The order in which the arguments are given do not matter. Every point in the space is zero distance from itself. For any three points, the triangle inequality holds.

The challenge is to arrive at a metric which can cope with comparing symbolic information. Literally, how similar is an apple and an orange compared with an orange and a tangerine? Intuitively, one pair is more similar than another, but the question is how this similarity can be expressed and even computed. The algorithms here try to accomplish this goal based on their worth in predicting the goal. They are from Stanfill and Waltz [Sta86] and Maes and Kozierok [Mae93] [Koz93]. Stanfill and Waltz used the algorithms on the NETtalk pronunciation problem [Sej86], and Maes and Kozierok have used them for computing electronic mail interest and intelligent meeting scheduling. As Stanfill and Waltz note, they can offer only little other mathematical motivation for their metrics, other than that it seems to work well empirically, so it may be possible to develop even better metrics.

7.3.4 Stanfill/Waltz distance metric

To introduce the metric, we establish some terminology and notation. For a situation vector, we make the distinction between *predictor* and *goal* fields of the vector—a predictor is a field whose value is known and used in the prediction of other fields, and the goal is the field whose value is to be predicted. Let D be the database of memorized vectors. Let ρ be a memorized vector and τ be the *target* vector containing goal field(s). We say $\rho.f$ is field f of record ρ. $[f = v]$ means that feature f has value v. Let $D[f = v]$ be the database restricted to those vectors whose field f has value v, and let $|D[f = v]|$ be the count of the restricted database.

Stanfill and Waltz use the following distance metric is used to compute the distance between target τ and a given situation ρ. Their formulas are implemented for the thesis:

$$\Delta^g(D,\tau,\rho) = \sum_{f \in P_\rho} \delta_f^g(D,\tau.f,\rho.f)$$

$$\delta_f^g(D,\tau.f,\rho.f) = d_f^g(D,\tau.f,\rho.f) w_f^g(D,\tau.f)$$

$$w_f^g(D,\tau.f) = \sqrt{\sum_{v \in V_g} \left(\frac{|D[f = \tau.f][g = v]|}{|D[f = \tau.f]|} \right)^2}$$

$$d_f^g(D,\tau.f,\rho.f) = \sum_{v \in V_g} \left(\frac{|D[f = \tau.f][g = v]|}{|D[f = \tau.f]|} - \frac{|D[f = \rho.f][g = v]|}{|D[f = \rho.f]|} \right)^2$$

where P_ρ is the set of predictors in the situation (excluding the goal field), f is a member of the set of fields representing the situation and V_g is the set of encountered goal values in D. If situations can have more than one goal field, then the metric will be different for different goal fields.

At the top level, the metric is a simple sum of the distances of the corresponding features in the target and the memorized situation. For each feature, the feature distance is the product of a distance and weight calculation. The motivation behind these formulas is that they attempt to measure how well each field predicts the goal field. The squaring operation and square root induce the distance conditions.

The weight calculation w_f^g is a measure of how well a given feature of the target predicts the goal and is a measure across the entire database of situations rather than a particular situation. It is computed by first restricting the database to the part for which the feature has the same value as the target's field. Clearly, if the database has no recollection of the feature's value, then its weight cannot be computed and degenerates to 0. Otherwise, it is computed as the root of the squared sum of the fractions of set sizes of the observed goals over the size of the restricted database. The net result is that the weight value is higher if there are fewer goal values given the target's feature value, and lower if there are more goal values.

The intuition behind this calculation is that some fields correlate strongly with the goal field because their values predict only a handful of goals (or just one goal in the strongest case), while other fields correlate less strongly. As a simple example, consider a particular user who always reads a message if it is addressed specifically to him, regardless of what the sub-

ject line says. In this case, the "address" feature strongly influences whether the user will read a message. In such instances, the given field should have a strong say in the distance computation because its value biases the goal heavily. For instance, if the encountered goals have only one value, then its weight is at the maximum of 1 and the feature weighs heavily in the calculation of distance. The subject feature does not influence the user's decision as much, so it should be weighted less in the distance measure. More "noisy" fields such as these have less predictive power when the field value yields a diverse set of goals, and they hence should be considered less strongly. The calculation of the feature weight encodes these intuitions.

The distance calculation d_f^g is a measure of how differently the target feature value and the memorized feature value predict the goal field. Unlike the weight calculation, this formula takes into explicit account the actual differences between the target's feature and the memorized feature. It is computed by summing squares of the differences of the two features in predicting the various goal values. The driving idea is that different feature values are equivalent if they end up predicting the same goal, so the fact that different values occur in the same field should not make the values "distant" if they are in fact "equivalent." It is easy to see this idea in the formula in the case where the feat $f = \tau.f$ and $f = \rho.f$ predict the same goal exactly the same number of times. In this case, the difference is 0 and so the distance of the two features is 0. This estimation can be analyzed as the combination of three steps iterated over all observed goals. First, the calculation pares away the above case where the target and memorized feature predict the same goal exactly the same number of times. This is the case of complete overlap of the two feature values, and no distance is added into the measurement. Second is the partial overlap case where both values predict the same goal but with slightly different frequencies. In this case, the difference in the frequencies are squared and added to the sum. Third, if one of the feature values predicts a goal value that the other feature does not predict, then its frequency of predicting that goal is also added to the sum. Clearly, if the frequency of occurrence of this "one-sided" goal is high, it will tend to add more to the distance measurement. This last case is the disjoint case. Since the difference is squared, the differences of the target field and memorized field are taken into account symmetrically.

For instance, suppose the user reads mail whether it's from Lisa or Barry. It can be said that if the sender is the only feature upon which to base a read decision, then the situations of mail from Lisa and mail from Barry are not really different (and hence not distant) because the user treats both situations in the same way. The distance formula accounts for these intuitions.

7.3.5 Computing a confidence measure

The above formulas are used to compute the distance values between the current situation τ and all records ρ in the database. One of the weaknesses in these formulas is that if no evidence is present for a field, its weight goes to zero and the field is not counted in the distance. This means that in the worse case when all fields have new values, the distance will be 0. Clearly, one must be able to distinguish between the case when there is an exact match and when there are absolutely no matches. A confidence value can help distinguish between these cases.

This value is predicted by first retrieving the m closest situations from memory for some predefined m. The algorithm always returns situations which match the target situation perfectly, or have 0 distance, so if more than m situations have distance 0 from the target, then more than m situations are returned.

The goal field of the situation with the highest "score" will be chosen as the prediction. The score for each action which occurs in the closest m memorized situations is computed as:

$$\sum_{s \in S} \frac{1}{d_s}$$

where S is the set of memorized situations predicting the given action, and d_s is the distance between the current situation and the memorized situation s. The algorithm selects the goal field with the highest score.

Kozierok uses the following formula to compute a confidence measure for the prediction. It is implemented as part of the MBR algorithm for the thesis:

$$\left(1 - \frac{d_{predicted} \,/\, n_{predicted}}{d_{other} \,/\, n_{other}}\right) \times \frac{n_{total}}{m}$$

where:

- m is the number of situations considered in making a prediction.
- $d_{predicted}$ is the distance to the closest situation.
- d_{other} is the distance to the closest situation with a different goal than the predicted.
- $n_{predicted}$ is the number of the closest m situations with distances less than a given maximum. A maximum is chosen to exclude any exceedingly distant situation from consideration. Note that this value may not equal m if more than m situations have distance 0 to the target.

- n_{other} is the minimum of 1 or the number of the closest m situations with distances within the same maximum with different actions other than the predicted one.
- $n_{total} = n_{predicted} + n_{other}$ is the total number of the closest m situations with distances below the maximum.

If the computed confidence is less than 0, it is set to 0. This situation happens in the case that $d_{predicted} / n_{predicted} < d_{other} / n_{other}$, which is usually the result of several different goal values occurring in the top m situations. If every situation in the memory has the same goal attached to it, then d_{other} has no value. In this case, the first term of the confidence formula is assigned a value of 1. However, it is still multiplied by the second term, which accounts for the number of experiences the database has collected. When the reasoner has very few situations, this second term will lower the overall confidence value.

Finally, a threshold T can be established so that if the confidence is below it, no recommendation to the user will be made. Above T, the prediction is suggested to the user, who accepts or rejects the prediction.[3] The value of T is domain-dependent.

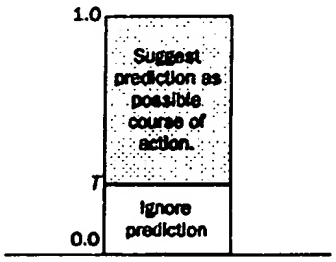

Figure 7.1: Threshold for suggestion.

7.3.6 Implementation

The memory-based reasoning algorithm is implemented as a generalized class called MBRStore, which accepts and performs reasoning on arbitrarily-sized feature vectors and data content based on the dtype data representation. (See "Data representation with dtypes" on page 42. A vector here is implemented as a dt_list dtype.) An MBRStore instance is initialized with the vector length of situations to be stored and a list of indices indicating the goal positions in the vector. Note that the vector length is fixed for a given MBRStore, so the feature set is also fixed and the corresponding locations in any two situation vectors contain values for the same feature.

3. Maes and Kozierok actually use two thresholds—one lower and upper threshold. If the confidence is below the lower one, the prediction is rejected outright. If it is between the two, the agent suggests it as a possible course of action. Otherwise, the action is taken on behalf of the user.

The nature of the algorithm exploits the parallelization possible with parallel processors for efficiency, but on serial machines it can run quite inefficiently. Besides storing the vectors directly, an MBRStore object also computes a cache as a situation is added. The cache is a four-level tree giving a breakdown of counts of features and their values, which the algorithm uses so often. The first level breaks down the possible goal fields; the second separates the feature fields; the third breaks down the encountered values of its parent feature, and finally the fourth separates the goal values for the parent feature value. The terminals of the tree give the count for $|D[f = \rho.f][g = v]|$, for any given f, $\rho.f$, g, and v.

MBRStore objects has a default storage limit of 100 situations but can be programmatically set to hold an arbitrary number of situations as system memory permits. When this buffer is exceeded, the oldest situations are purged from memory and the cache correctly updated. Instances of MBRStore are also persistent, so their memories are saved and restored across execution sessions.

7.4 Integrating Learning in Chatter

The main motivation for developing memory in an interface is to use it as a basis for making suggestions, and these suggestions must be fitted into the context of a discourse. Specifically, it means deciding what types of information and events are appropriate to make suggestions about, and then deciding when to do it. This section addresses these two points.

MBR is desirable in the context of interface agents because: (1) it can learn the regularities of a user's behavior; (2) more significantly, it can learn about the exceptions to those regularities. In this sense, MBR captures a desirable trait of user preferences not easily encodable using rules. Yet, the use of these terms may be misleading because regularities and exceptions are rather different ends of a user preference scale. They are distinguished only by number of occurrence out of an example base, and what may be an exception now may become the rule later. MBR captures these conceptions nicely by recording a set of instances.

However, the main challenge in using MBR is to come up with the correct set of features to represent the program's domain. If an inappropriate set of features is chosen, it leads the interface into making the wrong set of inferences about user actions, resulting in an annoyingly suggestive interface.

Since dialog segments organize the tasks of an application, one way to think about choosing the correct features to represent is to consider using learning to support the interaction within segments. The last chapter (see "Choosing appropriate segments" on page 59) dis-

cussed the organization of segments around an object of interest or a major application function. In Chatter, such an object of interest may be a message; a function can be sending mail. MBR can be used to support these functions on a per segment basis.

7.4.1 Collecting message features

One approach to making the interface more active is to have it bring up interesting information to the user. Presumably, if the user has nothing to talk about, the interface can always strike a conversation with a new topic. In Chatter, the objects of interest are e-mail and voice mail messages. A list of messages sorted from "most interesting" to "least interesting" can be constructed by the agent to make conversation with the user when he is asking the interface for new topics of discussion. These rankings can be based on how much of past, similar messages the user has read. This section discusses how the sorted list is computed. The next section describes how it is presented. Ranking information according to interest in the audio domain is more important in the audio and portable domains, where time and presentation space is more limited. The user has limited time, so it is important to present the most relevant information first.

MBR can be used to generate a sorted list of messages. The following features are collected for deciding the interest level of messages:

Type	Features
E-mail	Subject Is it a reply message? Sender Recipient Is the mail is directly address to the user? Mail domain of sender Quantized length (5 values) Interest flag
Voice mail	Sender string Does the call have an associated caller id? Quantized length (5 values) Interest flag

Figure 7.2: Message features for MBR.

Some of the features are directly copied from the message's header fields and others are derived properties of the message.[4] The *quantized length* feature is an integer value repre-

4. Note that the message body is not analyzed for interest content. This in general is much more difficult to do.

senting a description of the length of the message. The actual length, measured in number of characters or number of seconds, is not used because it is somewhat meaningless in the MBR algorithm. Collecting lengths into bins allows the MBR algorithm to work better at producing matches for the length field, so the actual length is reduced to a value from "very short" to "very long". The *interest flag* feature deserves special mention. This flag is not actually a feature of the message but of the situation during which it was read. The flag is true when the user has attempted to contact the sender by mail or phone through the interface. This event is considered important because presumably, the user is attempting to communicate with the person, and any messages from that person may have to do with communicating with him. One of Chatter's aims is to facilitate communication between users. This flag is reset to false when the user replies to one of the sender's messages or is able to call him through Chatter.

Since the system is designed not to be programmed by users, the feature set attempts to encompass many the factors that any user might implicitly use in deciding whether to read or listen to a message. The choice of features also descends from informal observations of rules that users in the work group used for filtering their mail with the *Procmail* program. It was found that all *Procmail* users based their interest level of particular messages on the content or state of one or more of these features.

Once the features of messages are represented, the interest level the user gives them must also be represented as part of the situation. This table presents the four possible levels and how user actions are translated into one of the actions:

Interest level	How it gets this label
Very interested	User listens to message on first pass or interest flag is on.
Interested	User listens to all or part of message on a subsequent pass.
Not interested	User does not listen to message body at all.
Ignore	Upon hearing sender and subject, user deletes message without listening to message.

Figure 7.3: Interest levels of messages.

When Chatter presents messages, it does not read message bodies by default on the assumption that most messages will be either uninteresting or too lengthy. The interface only recites the sender and subject of each message. In the table, *first pass* means that the user

chooses to listen to a message when it is presented to him the first time. *Subsequent pass* means he reads the message when re-scanning the message list.

Statistics are collected at the end of an interactive session with Chatter, and the sort for new messages will occur at two levels. The list is ordered using the four interest levels. When a new message arrives, its interest level is predicted using MBR, which places the message into an interest category. The confidence level of the message is then used to rank the message inside the category. This process is used to generate a sorted list of messages.

Here, MBR is used mainly for filtering information, a side effect of which is used to drive the agent's suggestive behavior. This list can be global across interface media (e.g. on-screen or over the telephone). The assumption is that user's interests will be mostly consistent no matter how he gets the information. The presentation will clearly differ across mediums, but the point is that the list is always globally ordered.

7.4.2 Introducing message suggestions

Bringing up suggestions when the topic is not directly relevant results in a fragmented experience for the user. Once suggestions are collected into a buffer, they must wait until there is an appropriate point in the dialog for changing the topic. Ma l, as well as other events, can arrive asynchronously during an interactive session, but if the computer made an alert right away, the interruption may be distracting if the user is not already reading mail. Another issue to consider is whether the user or agent is controlling the interaction. While the user may want the agent to bring up suggestions, he may at other times want to initiate his own actions, such as to quickly send a message. Some means must be provided for sharing turns in taking initiative in the conversation.

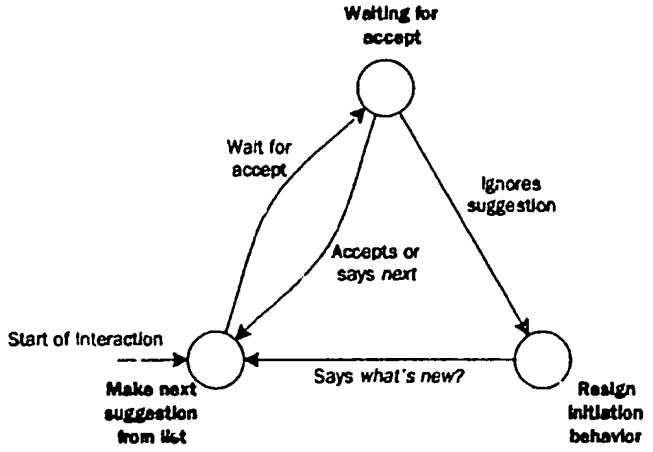

Figure 7.4: State machine for taking and resigning initiative.

As illustrated in figure 7.4, Chatter maintains a list of interesting topics and implements the following state machine for taking and resigning the initiative. When the user first begins a dialog, Chatter attempts to take the initiative by making suggestions about interesting messages. This process continues as long as the user is interested. When the user ignores a suggestion, Chatter goes into a state where it does not suggest new information of interest until the user asks (by saying *what's new?*).

7.4.3 Suggesting actions

Suggesting actions for the user to take is distinguished from suggesting information because such suggestions are application-specific. As applications become more feature-ladened, users will have an increasingly difficult time remembering all its possible commands. The fully user-directed interaction style so common today breaks down. To make applications easier to use, some interfaces take the "guided" approach, in which at the end of every step or command, the interface suggests another step deemed appropriate by the application's designer. The intuition is that performing one task might suggest another task. For example, when someone reads a message, the next step may be to reply to it.

To see the problem with this and other "hard-wired" approaches, one must look at the problem more abstractly. An application can be viewed as a graph of capabilities and tasks. The tasks make up the nodes of a graph and the arcs in the graph represent a logical progression of tasks designed into the program:

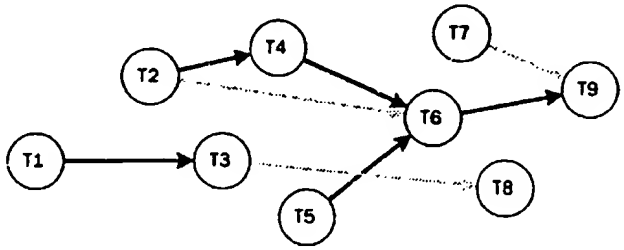

Figure 7.5: Abstract task space.

For example, T2 may be the task of reading a message, T4 represents the task of replying to it and T6 is the task of filing the message away. Clearly, different data can cause different node traversals, so the graph is often labeled with different-colored arcs. The gray arc from T2 to T6 represents the suggestion of saving the message without replying. In this model, a complex task is performed by stringing together individual tasks in some coherent order, or making a partial traversal through the graph.

One basic approach to organizing these tasks is to structure the program's various com-

mands into a menu tree. Making a suggestion in the course of an interaction then becomes easy; the computer only has to present a menu of choices and the user selects one he wants. There is one large drawback to this approach, namely that navigation to other places in the tree is tedious. A somewhat better method is to organize a program's capabilities into some loosely unordered set as mentioned above, and then invoke an interface designer's expertise in choosing the appropriate guided task progression. However, what if there are several appropriate task progressions? The choice may depend on user preferences or even the users' skill level. The application designer is now forced to choose the task that most users will want to follow. In the speech domain, it's even more important not to spend time making inappropriate suggestions because any time spent on unwanted choices is wasted time. Programs can perform more effectively by knowing user needs.

MBR provides the mechanism for building an emergent task structure. Initially, the user begins with an application with a somewhat unstructured task set. As the user uses the application more and more, the transitions from task to task are remembered. Connections between tasks are established, and the interaction becomes more guided, except that these links are customized to the individual. The interactive experience becomes more efficient because the agent can suggest an appropriate course of action to which the user needs only to answer *yes*, or give another choice to proceed. If no regularity can be found between two tasks, then the MBR algorithms return a low confidence level and no suggestion is made to the user. In theory, any "hard-coded" interaction decision can be learned. However, in practice, care must be taken in introducing MBR at the appropriate possible transition points in the interaction. Much of the time, tasks will assume a natural, logical order, and it can be explicitly coded into the program. In other cases, the progression may be less clear and user preferences will vary.

These ideas have been used in Chatter most deeply in the segments related to reading mail. After a message is read, one or more of the following actions can be taken on the message:

Action	How it gets this label
Reply	User replies to message.
Forward	User forwards message to another user. (In a forward, the recipient is also recorded.)
Save	User saves message into mail folder.
Delete	User deletes mail after reading it.
None	User reads next or previous message.

Figure 7.6: Suggesting an action.

As an action is taken on an individual message, it, along with features of the message, are recorded in the database for use in predicting future actions. The actions above are not all mutually exclusive, so if more than one action is taken on a message, then the system simply assumes that they are all likely candidate actions and asserts records for each action taken.[5] When the agent has a high confidence on a suggestion, it asks whether the user would like to perform the task, and the user only has to answer with *yes* or choose to perform about task. If the agent's suggestion is rejected, the new choice is added to the memory. With a low confidence, the interface simply waits for a user command.

Other segments have simpler preference modeling, as their functionality is more limited. The features used in representing situations for other segments are given here:

Segment	Features	Predicted Actions
BaseSegment	None	Most likely invoked segment?
CallSegment	Called party's name	Call work or home number?
ComposeSegment	Recipient's name Action after mail is sent	Use what delivery means? Action after mail is sent?
ExitSegment	None	None
GreetSegment	None	None
HoldSegment	None	None
PersonSegment	Information requested (e.g. phone number, address) Action(s) taken on person	Information requested next? Next action?
ReadSegment	E-mail and voice mail features above Action(s) taken on message	User interest in message? User action on message?

Figure 7.7: Features of situations.

Separate MBRStore objects are used to store features per segment class. For the BaseSegment, no features are used, yet an action is still predicted. This prediction is the segment that is most likely to be invoked when the interaction has no topic of conversation at hand. Intuitively, the system tries to bring up a previously interesting topic for discussion.

One weakness of this learning scheme is that the agent does not have any meta-level knowledge about a series of tasks constituting a higher-level task which can be more conveniently

5. This assumption is probably not the best one, since it has some undesirable consequences for prediction.

labeled as a task. The interface cannot therefore offer to automate an entire string of tasks because it has no language for expressing such a string.

7.4.4 General scheme for interaction

While machine learning makes an interface malleable, it is still no substitute for correctly encoded domain knowledge about the user. For example, if a database contains information about the user being able to read voice messages, then whatever preferences the system learns about the user should be weighted less importantly. However, this information is also no substitute for the preferences the user's expresses in a command, which may contain information overriding the encoded preferences. Thus, there are distinct qualities of knowledge.

In obtaining information, Chatter's dialog segments are generally implemented using the following algorithm, in which learning is only one way to make suggestions:

1. **See if the information is already provided as part of the instructed command.** For example, if the user says *send a message to Don*, then the recipient is already given.
2. **Whenever possible, look in databases for information.** The Chatter domain provides some opportunities for retrieving information about users from pre-programmed databases. Some arguments for commands may be filled in by information from the database based on already provided values. For example, a database exists for users of voice mail subscribers. If the user says *send a message to Barry*, and Barry is voice mail subscriber, then suggest sending Barry voice mail.
3. **Determine whether MBR can guess at some of the unknown features of the situation.** A partially instantiated situation provides useful context, which can be used to predict some of the unknowns of a new situation. If so, then suggest the most-likely feature value. The user can a chance to accept or reject the suggestion.
4. **Engage in a dialog with the user to query for the necessary information.** If the interface cannot find the information anywhere, it can only ask the user to supply it.

7.4.5 Adapting to changing user preferences

As the preferences of the user change, the memory has the ability to adapt by "forgetting" the oldest instances. As described above, the memory has a storage upper limit, which when reached results in the purge of its oldest situations. Forgetting has the advantage of keeping the memory tuned to changing user preferences. Past, unwanted habits are no longer kept, but desirable habits are reinforced in memory by frequent assertions in the database. Such adaptation is useful in a mobile setting facilitated by Chatter, where the user may have

changing information needs. For instance, when the user travels, his information interests may change because he may be more interested in messages from his place of travel. The disadvantage is that forgetting can also result in the loss of rare but useful situations.

One way to fix this problem is to develop some methods which delete instances from the memory based on their usefulness in prediction rather than age. The system may reward those situations which have matched new situations more often, and delete the ones which don't seem to be relevant any more. [Koz93] suggests a method whereby older situations are gradually decayed in importance, which may result in better performance than a simple, sharp cut-off. However, this problem does not seem to arise in actual usage of Chatter.

8. Performing Recognition

This chapter discusses the issues in designing Chatter's speech recognition subsystem, its grammar design and the methods used for subsetting the grammar to gain better accuracy. It discusses the issues involved in using a speaker-independent, connected-speech recognizer. It also describes the design and implementation of a recognition server based on an internal object-oriented engine API.

8.1 Using Recognition

Chatter uses a speaker-independent, connected-speech, large vocabulary recognizer from Texas Instruments [Whe92]. In an ideal setting, speaker-independent recognition worksrobustly and the speaker's utterances are perfectly translated into text. This ideal setting makes the work of discourse management much simpler. Yet, in reality, recognizers are unreliable devices, susceptible to background noise and recognition errors. From an interface design standpoint, these difficulties diminish some ease-of-use advantages associated with speech. The user is troubled by not being understood, or worse, the computer blindly proceeds to do something other than what the user says. Errors can be minimized by incorporating higher-level knowledge to limit the possible utterances and therefore recognition errors. Below, methods used in Chatter for constraining the speech space are discussed.

8.1.1 Constraining recognition using a grammar

The recognizer used by Chatter uses a context-free constraint grammar for specifying and constraining the space of speech that can be recognized. This is also common practice for recognizers with large vocabularies that use Hidden Markov Modeling algorithms. The grammar is loaded in by the recognizer at initialization, and recognition is performed on it subsequently. The advantage of this approach is that accuracy and speed are greatly improved over an unconstrained system, such as IBM's Tangora System [Jel85]. These two reasons primarily motivated the use of the approach in Chatter. The disadvantage is that the possible speech input is constrained, so the recognizer is predisposed to the possibility of misunderstanding speech not defined as part of the grammar. This limitation applies not only to sentences not syntactically represented but also to words not among the grammar's terminals, the vocabulary. The recognizer always returns a correct output according to the grammar it is given, albeit with a confidence measure, but it is prone to severe errors if the user's speech is outside of the grammar's space. The implication for conversational systems is that error detection and repair become more difficult.

Specifying a conversational grammar involves a tradeoff between generality and accuracy. Clearly, one driving motivation is to write a grammar that is as loose as possible so that the same thoughts can be spoken in different ways. However, generality makes the recognizer less accurate because the space of possibilities, and hence error, is larger.

The best grammar to use is one that is as tightly constrained as possible. A grammar, specified in a straightforward natural language processing style with standard syntactic categories like noun phrases and verb phrases, causes over-generation of sentences. For example, the utterance *play an reply for Don* is syntactically correct but does not much sense in the context of Chatter. As a result, substitution errors are many and processing speed degrades. The recognizer tends to return utterances which may be syntactically correct but is in fact semantically non-sensical. One common solution to this problem is to encode semantic and even discourse constraints into the grammar, so that implausible sentences cannot be recognized[1]. In Chatter, semantic information is encoded into the recognizer's grammar. For instance, there are several commands in Chatter's vocabulary having to do with mail messages and people; one of them is for sending mail. Composing a message requires giving the kind of mail (e-mail or voice mail) and the recipient's name, but this information can be given optionally and perhaps in a different order. Simultaneously, some objects cannot be mailed, like *phone* or *company*, so they should not be specifiable in a send command. The following grammar encodes these semantic constraints:

```
ComposeMsg ---> ComposeCmd.
ComposeMsg ---> ComposeCmd AMailObj.
ComposeMsg ---> ComposeCmd AMailObj Prep Person.
ComposeMsg ---> ComposeCmd Person AMailObj.
ComposeCmd ---> "record".
ComposeCmd ---> "send".
ComposeCmd ---> "mail".
AMailObj ---> OptAMODet "email".
AMailObj ---> OptAMODet "email" "message".
AMailObj ---> "mail".
AMailObj ---> OptAMODet "message".
AMailObj ---> OptAMODet "voice mail".
OptAMODet ---> "an".
OptAMODet ---> "a".
OptAMODet ---> "".
```

Figure 8.1: Tight grammar for sending a message.

To send a message to Gayle, the user can say *mail*, *record a message*, *send e-mail to Gayle*

1. A better solution is to incorporate as much semantic and discourse knowledge as possible into the lowest levels of the recognizer. However, given that the recognizer used only allows the specification of a context-free grammar, we must implement these mechanisms above the recognizer. See [Hir91] for a description of the so-called *N*-best approach, which combines syntax and semantic constraints in guiding recognition. In addition, [Sen91] proposes some ideas on using discourse knowledge to improve recognition performance.

or *record voice mail for Gayle* but not anything non-sensical. Other commands optionally require either a person or a mail type, so the arguments for these commands are given specifically for them. See page 107 for a complete listing of the grammar. Encoding discourse constraints means selectively activating sections of a grammar according to what can appropriately be said at any point in the discourse. See the next section for discussion of this topic.

Unfortunately, a tightly constrained grammar is not without disadvantages. It tends to be much larger in length and more difficult to extend. Yet, recognizers perform most efficiently in accuracy and speed when the grammar given to them is tightly constrained.

8.1.2 Using start hypotheses

Recognizers can be dynamically programmed to selectively enable grammars on which to perform recognition, and this facility can be used to further constrain the grammar based on discourse constraints. Subsetting the grammar increases efficiency and accuracy. In Chatter, the entire grammar does not need to be active at all times; only the sub-grammar for the currently active dialog segment and the sub-grammars necessary to introduce or reintroduce inactive segments are activated. Grammars are individually enabled by the use of *start hypotheses* in Chatter's recognizer. They roughly correspond to the top-level non-terminals of grammars. At any time, one or more start hypotheses may be active. In Chatter, a hypothesis called Default is always active. This hypothesis contains all the commands that are active at any time, including calling a person, sending a message, reading messages or logging off.

However, some commands only make sense in certain contexts. They should be deactivated whenever possible until they can be used. To continue to above example, forwarding a received message to another user without a possible message reference is not appropriate, so the forward command is activated until this situation holds true. A Read hypothesis is defined containing the forward command, along with others. It is activated along with Default when the user is reading messages.

Chatter's grammar has seven start hypotheses: Default, Read, PersonInfo, ComposePerson, ComposeType, ComposeFormat, Ack. They are selectively enabled based on the state of the discourse. The decisions for activating and deactivating hypotheses are application-dependent, and they are implemented by the application programmer.

Start Hypothesis	Things the user can say
Default	Top-level grammar. Users can initiate any Chatter task with this grammar.
Read	Allows the user to manage his messages once he begins reading mail, such as replying, saving, deleting and so on.
PersonInfo	Once the user has activated a PersonSegment, this grammar allows the user to ask questio - about the user, such as *what's his phone number?*
ComposePerson	Activated in the ComposeSegment to accept a fragment as a response to a question such as *send it to who?* It accepts responses like *to Barry* or simply *Barry.*
ComposeType	Activated in the ComposeSegment to accept a fragment answer to the question *send him e-mail or voice mail?*
ComposeFormat	Activated in the ComposeSegment to accpet a fragment answer to the question *send it in what mail format?* The answer is any one of the supported multimedia mail formats.
Ack	Lets the user answer *yes* or *no* questions posed by Chatter.

Figure 8.2: Recognizer start hypotheses.

8.1.3 Enabling start hypotheses and trading accuracy

One important design issue to consider is the tradeoff between enabling multiple start hypotheses and losing recognition accuracy. As more and more start hypotheses are enabled, performance is decreased. This problem is even more significant in the context of nested and interruptable discourse, for a nested discourse must have the vocabulary for its active segments activated. An interrupted segment needs to be capable of being reintroduced at any time, so its vocabulary subset must also be enabled. If many segments have been interrupted, then many vocabulary subsets may also be enabled, and the recognizer loses the gains from subsetting. One solution is to create a minimal subset for each segment's vocabulary so that when a segment is active but suspended, the vocabulary for reintroducing the segment is active. When the suspended segment is reactivated, it can be

activate a larger vocabulary. In actual use, segments may not be interrupted very often, so this method may not be required.

Chatter uses the simple method of enabling the vocabulary for the currently active segment and all segments which have been interrupted so that they can be reintroduced at any time.

8.1.4 Segregating sexes

Chatter's recognizer generates separate male and female phoneme models from the lexicon for recognition, so an additional method for constraining grammars is to segregate users into sexes. Performance gains can be obtained by knowing whether the user is male or female and by using this information to constrain the grammar. Often, speaker-independent recognizers, such as the one being used for Chatter, will automatically employ both male and female phoneme models to more closely match the speaker's speech. Applications having user accounts can also record the sex of individual users, as it may be entered as part of configuring the system for a new user. The advantages gained by this method is that it eliminates half the hypotheses necessary for testing, which increases both recognition accuracy and speed. As soon as the identity of the user can be established, the recognizer can be told to use only phoneme models for one sex. (See [Hir73] for an interesting discussion of female-male differences in conversational interaction.)

8.1.5 Interrupting speech output

Besides being able to recognize speech reliably, a recognizer also performs the role of knowing when to stop talking. In conversation, speakers often interrupt each other when the point of the other speaker's utterance can be guessed. When speakers cannot be interrupted, interaction seems slow and tedious. It is essential to allow interruptions in conversational systems to make them responsive. For instance, while being read a long message, the user may want to interrupt to perform another task. The recognition subsystem should have some mechanism for detecting the beginning of the user's speech, so that it knows when to stop speaking and listen to the user's command.

Detecting speech in an audio stream can be done, but the problem is exacerbated using an analog telephone interface such as the one being used for Chatter, where the audio of both speakers are combined into one channel. Performing the user's speaking is more difficult because the DECtalk's voice must be separated from the rest of the audio. To overcome this problem, Chatter can be currently interrupted by the user pressing a TouchTone. TouchTones can be reliably detected and Chatter will stop speaking and wait for the user's command.

This problem can be solved by using an ISDN telephone interface at the computer, as ISDN provides separate data channels for both speakers. Detecting speech can then be performed on the incoming channel only.

8.2 Recognition Server

Recognition for Chatter is handled by a recognition server, the design and implementation of which is discussed in this section. The key motivation for having a server are its abstraction advantages. Recognizers of today differ in capabilities and programmatic interfaces, and developing applications with recognition functionality means writing to non-portable protocols. As new, improved recognizers become available, such applications will need to be rewritten for these new protocols. A recognition server can abstract such differences from client applications such as Chatter, while also supporting the use of different classes of recognizers available today. Like window system servers, recognition servers provides a suite of functionality based on an application's needs, while allowing a device-independent means of accessing such functionality. Such a server can also offer multiple client applications the simultaneous ability to perform speech recognition functions from the audio stream. A server can also offer a range of functionality depending on a given application's needs. Besides the speaker-independent, connected-speech recognition functionality used for Chatter, the server also supports speaker-dependent, isolated-word recognizers.

The current design of the server already incorporates three recognition engines—from TI, Agog, and Command Corp. The TI recognizer offers speaker-independent, connected-word, large vocabulary recognition; the Agog offers speaker-dependent, isolated word and small vocabulary recognition, and the Command Corp recognizer, called IN^3, offers speaker-dependent, keyword recognition.

8.2.1 Design Issues

For the server to be general enough to accommodate most kinds of recognizers, it must deal with both similarities and differences of recognition technologies available today. The main differences among recognition technologies can be represented along the three axes of speaker-dependence, connectedness of the recognition, and the vocabulary size.

Each point in this space can be considered to represent a particular class of recognizer. For example, it may be possible to have a large-vocabulary, continuous, speaker-dependent recognizer. Each point is also distinguished by its own operational requirements; some recognizers, such as the one used for Chatter, require language grammars while others require the training of word templates. The server must therefore provide provisions for clients to

select the kind of server they want, and furthermore to relay any additional data between the engine and client necessary for successful operation in a device-independent manner.

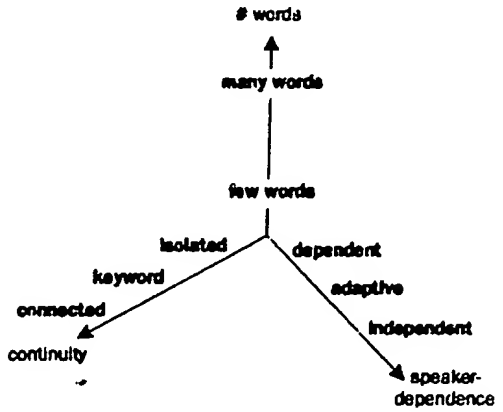

Figure 8.3: Recognizer functionality space.

Although each point is a possible class of recognizer, current recognizers fall into roughly three categories:

- **Speaker-dependent, isolated-word, small-vocabulary.** Recognizers in this class are perhaps the simplest in implementation. Users of such recognizers supply training samples by saying each word[2] of the vocabulary several times, and these samples are used as templates in matching speech at the recognition stage. A special kind of recognizer in this class includes the word-spotting recognizer, which attempts to pick up all or any keywords in a specified vocabulary from a continuous speech stream. Such recognizers may be more useful in certain domains where commands may be given as complete sentences, in which case they can be used to pick out the essential information without much regard to analysis of syntactic structure.

- **Speaker-independent, isolated-word, small-vocabulary.** These recognizers are good for simple recognition tasks where accuracy and widespread speaker applicability is essential. Perhaps the most prevalent use of these recognizers are with over-the-phone consumer services, where particular items from a menu, a binary choice, or even a digi ce to be spoken by the caller. Implementation of these recognizers borrows from the previous class of recognizers and the class of connected-word, large-vocabulary recognizers, to be discussed next.

- **Speaker-independent, connected-word, large-vocabulary.** This class of recognizers rep-

2. The term "word" is used fairly loosely here, when in fact, almost any kind of repeatable sound, including a series of words, can be accepted on recognizers of this type.

resents the state-of-the-art in current recognition research. Word and/or speech samples are collected from a large set of speakers and statistics of word frequencies and phoneme characteristics are distilled. Such recognizers typically use some form of Hidden Markov Models on a constraint grammar as the basic recognition algorithm.

It is useful for the server to provide roughly three sets of functionality for clients. To accomplish these ends, the architecture of the server is constructed in an object-oriented and inheritance-based manner. The motivation for organizing the server architecture in this way is the observation that first, it must provide a common set of functionality across all specific instances of recognizers, which implies that its implementation must make up for deficiencies of particular engines and also defer to those which provide their own (or better) means of performing part of the recognition task. Secondly, while all engines of a particular class are different in how they are accessed, they are in many ways structurally the same except for minor differences. Thirdly, different classes of recognizers may have significantly different methods of usage. For example, a recognition result from a connected-word recognizer may be an entire string of words, but an isolated-word recognizer may consist of only one word.

8.3 Server Implementation

These observations lead to the use of an object-oriented paradigm, in which the core functionality can be embedded into an abstract **Recognizer** root class, and real recognizers are implemented as subclasses of this root class. Subclasses can inherit functionality from the root that they can't provide, and it can override functionality if its own is better. More often however, subclasses will want to inherit the core functionality but will also provide their own to perform additional initialization or processing. For different classes of recognizers, abstract subclasses of the root class can be specified to provide a separate set of core functionality associated with that particular class of recognizers. All of these concepts can be represented naturally in an objected-oriented, inheritance hierarchy.

To enable the use of different brands of recognizers in the server, an internal engine API is also defined to ease the process of incorporating new recognition engines into the server. Future providers of recognizers can also give programmatic interfaces that make their incorporation possible more straightforward. As recognition engines with better performance become available, they can be added to the server, via this standardized engine API, and automatically become available to all applications which use the server without modification.

An abstract Recognizer class forms the root class. It implements functionality common to

all recognizers, regardless of their type, such as the mechanisms for handling connection control with the client, device-independent means of receiving audio data, converting between the necessary audio formats, loading and saving of data files, and the maintenance of state of the recognition process. Then, abstract subclasses of Recognizer are defined for each of the three common types of recognizer:

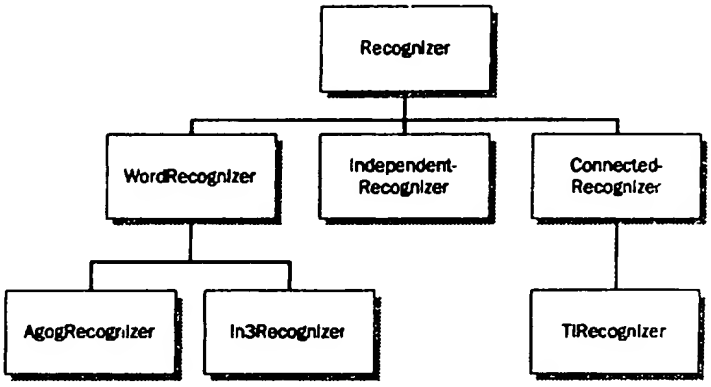

Figure 8.4: Internal recognizer hierarchy.

Above, the **WordRecognizer** class implements the speaker-dependent, isolated-word, small-vocabulary recognizer; the **IndependentRecognizer** implements the speaker-independent, isolated-word, small-vocabulary recognizer, and the **ConnctedRecognizer** a speaker-independent, connected-word, large-vocabulary recognizer. A subclass of **WordRecognizer** or **ConnectedRecognizer** serves as "glue" for the recognizer and server. The implementations of the actual recognizers are also shown. Currently, the **IndependentRecognizer** class is not implemented.

The various subclasses implement the protocol necessary for the operation of that recognizer class. For example, the **WordRecognizer** class specifies that training is to begin a message to **startTraining**, a **train** message to train individual words and finally a **stopTraining** message to end training. The **AgogRecognizer** simply overrides these calls to prepare its own internal voice models for training. To receive audio data, the **TIRecognizer** overrides a **receiveAudio** method inherited from **Recognizer**. **receiveAudio** is automatically invoked with audio data when the audio stream is turned on for recognition or training. More specifics are detailed in Appendix A.

Each connection to the server is represented as an object instance of a particular recognizer class, and calls to the server are simply translated to messages to the appropriate objects. The engine API is therefore specified as a set of member functions of the abstract Recognizer superclass.

8.4 Interfacing to Audio

The server is written to cooperate with an audio server [Aro92], which allows several applications to use the audio device as a shared resource. The server therefore serves as just another client of the audio server, although its processing is also useful to other applications. This diagram illustrates how the recognition server fits into the general audio server framework:

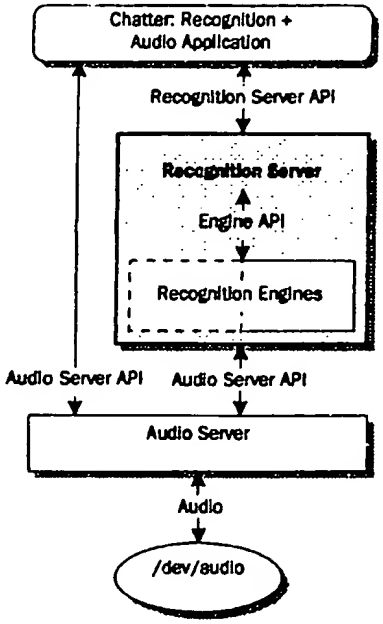

Figure 8.5: Audio application architecture.

An application wanting recognition functionality can communicate with the server to receive recognition "events" found in the incoming audio stream. It can also initiate the training of word templates to be used for later recognition. While the server serves as intermediary between the client and the audio server, a client application can still receive and send data directly to the audio server directly. This framework allows applications to perform mixed-mode audio tasks, where training, recognition, record and playback can be arbitrarily intermixed during their execution.

8.5 Interfacing with Recognizers

When designing programmatic interfaces for a specific recognizer, the designer should realize that his recognizer will be working in a shared-resource, asynchronous setting. This section outlines the key issues to be considered in designing such an interface and how the internal server architecture facilitates the integration of new recognizers. To avoid ambigu-

ity, the term *recognizer* refers to a recognition engine, while the term *server* will refer to the infrastructure of the recognition server. The client-server architecture is very similar to the one shown in Figure 4.4 on page 45.

8.5.1 Receiving audio

A recognizer must work in an environment where the audio stream is a shared resource, so it must be designed to not require unique access to the audio input but be able to accept audio from an arbitrary source. Several methods exist for transferring audio to recognizers in the recognition server. We discuss the two ways in which the server supports the delivery of audio.

One way is for the recognizer to receive audio from a named pipe. The recognizer and recognition server can establish a shared pipe, and when audio begins to arrive at the server, it can be written to the pipe and received by the recognizer for processing. This method was used to deliver audio to the IN^3 recognizer. It is most straightforward if the recognizer is already designed to read from the real audio device, in which case it is easy to change the device's name. The drawback of this method is that it is not very efficient because of the additional overhead involved in context-switch time to transfe · data using pipes to the recognizer.

A more efficient means is to pass audio to the recognizer with a predefined function call in the recognizer's API. The recognizer's API defines a function for receiving audio with arguments to specify the location and length of data, and when the server receives an audio buffer, it can deliver this buffer using the function. This method was used to provide audio to the Agog and TI recognizers. Note that it is the responsibility of the recognizer to manage the received audio thereafter, since the server passes only one copy of the stream to the recognizer. The recognizer can either process the new buffer as soon as it is received or save it in private buffers until the end of a speech segment is recognized. In the server's current design, audio will be lost if the function takes too long to return, so recognizer's API design should take into account the time it takes to process a frame of audio. However, providing a call to receive audio is the most efficient means of delivery.

Since delivery of audio can be suspended and restarted as dictated by the external functionality of the server, the recognizer should also provide a reset function which initializes its internal state for a new session of recognition. The problem is that the server sends audio to the recognizer only when either recognition or training is active, and the recognizer may receive a buffer of audio that is time-discontinuous from the previous buffer it received. If the recognizer performs recognition of these buffers as if the audio in them were contiguous

in time, it may incorrectly report spurious recognition results. A reset function provides the means to restart the recognizer in a clean state.

8.5.2 Posting recognition results

As a recognizer receives audio, it must somehow post recognition results to the server so that client processes may be notified of recognized speech. Since the server operates asynchronously, reporting results from recognizer to server must also occur asynchronousiy. Here, there are three methods for recognizers to post results: the server polls for a result; the recognizer reports results directly from the function used to pass the recognizer audio; or the recognizer invokes a callback to report the result.

In the first method, the server periodically polls the recognizer to check for any recognized events. The server may call a function or check a variable's value from time to time to determine whether any results can be obtained. This method is used to find recognition results for the IN^3 recognizer. The testing interval may be controlled by a timer or some other periodic event. Since delivery of audio data is a periodic event, the server currently checks for recognition results after several buffers have been delivered if this recognizer is being used. The drawback of this approach is twofold: processor time is wasted on polling, and there may be a lag time between when the recognition result is known and when it is discovered, which may affect the interactivity of the system. However, this method may be used for recognizers that weren't originally designed to be operated asynchronously.

A more efficient method is to return recognition results as the result of the function used to pass in audio. This method also assumes the particular style of audio delivery described above, and it is in fact used to determine recognition results for the TI recognizer. In this case, the function result can easily be checked to determine whether something was recognized. No result may be signaled by a null result. While this method removes the inefficiency of busy waiting, it may still result in lag response time if the audio buffer size is large.

The best method is to use a callback mechanism. The server registers a callback using a function in the recognizer's API. When the recognizer computes a recognition result, it immediately notifies the server, which also sends the client a recognition event. This method is used to obtain recognition results from the Agog recognizer, and it avoids the disadvantages of the first two approaches.

9. Conclusions

This thesis has described a conversational agent called Chatter that assumes a more active role by making suggestions for appropriate courses of action. The two main areas this thesis addressed are: (1) discourse modeling and (2) learning to make suggestions through user observation.

9.1 Distinguishing Chatter from Other Speech Systems

Chatter is distinguished from other speech systems by its two main features:

- Discourse modeling. Speech systems are still mostly command-driven, yet natural conversation is more subtle and complex because it takes context into account. Chatter models conversation more closely, allowing users to make references to past objects of discourse and interrupt tasks to perform others.
- Making suggestions. Chatter's interaction style is guided or agent-oriented rather than directly manipulated. It brings to the user's attention interesting messages and offers to automate some of his actions. It asks questions when it needs information to perform a task. As applications become more conversational and complex, there is an increased need for an interface to become active in suggesting appropriate capabilities to the user.

  User preferences are different, so to make good suggestions, the interface must learn what individual users like and use its memory as a basis for making suggestions. Chatter relies on a machine learning approach to collect relevant features about situations for each individual user and, in future situations, makes suggestions based on the assumption that the user will make similar choices.

9.2 Lessons and Issues

This thesis has concentrated on the study of the following issues. Lessons learned from this research also raise issues for future study.

- Portable information base. When information is shared among several input and output channels, its presentation must be coordinated so that it can be consistent. The consistency applies not only to the information itself but the filtering that is done on it. This thesis implements the necessary infrastructure in the form of an event server to permit information to be presented to the user no matter how he connects to his data.

- **Multiple ways to say the same things.** Natural language is easy to use because one thought can be expressed several ways. When designing conversational systems, one must also try to accommodate this multiplicity. The trade-off is between time and accuracy, but such sacrifices need to be made so the interface can work more effectively.

- **Accepting fragments.** Conversational speech is often disjointed or incomplete. The implication for the design of conversational systems is that they be able to account for fragmented utterances. Chatter's grammar is designed so that fragmented utterances are accepted for questions that require only one- or two-word answers. In other contexts, because of the way Chatter's recognizer works, partially complete sentences cannot be detected, so this information can sometimes be lost. Assuming this fragmented information can be recovered using other recognition systems, how can a conversational learning system learn deal with them? For example, if the recognizer only recognizes the word *Barry*, is it possible for the system to make a suggestion based on this scant information? Learning methods may be used to determine what the user wants to do at the mention of a name.

- **Taking turns.** As interfaces become more active in making suggestions, it must develop sensibilities for knowing when to speak up and initiate conversation and when to listen and let the user control the course of the interaction. This thesis has presented a state machine for turn taking, but it is only applicable to the topic level. How can methods be developed for detecting changes in the interaction where the user wants to be active or reactive?

- **Resolving pronouns.** In general, pronoun and anaphoric resolution is difficult to accomplish without a great amount of domain knowledge, yet it is one of the most useful devices of conversation. Chatter offers some simple mechanisms for using pronouns in discourse. What happens when the discourse domain becomes more complex, and several objects can be "juggled" at once, as in *put this in that*? How can domain knowledge be easily encoded so that users can make such references?

- **Error repair.** Chatter provides simple commands for repairing misrecognized utterances. The system echoes feedback on all information that a user gives so that the user knows whether he's being heard correctly. How can the condition be detected the other way around? What if the computer suspects that the user has been misled by the way he is answering questions? Furthermore, how can interaction methods be developed for unraveling preconceptions generated a few steps ago?

- **Supporting discourse through learning.** One of the main themes of this thesis is that human-computer discourse can be supported through the use of learning, for it provides the computational power for driving the computer's discourse. Chatter integrates

learning at two levels of discourse: suggesting topics—in this case messages, to be heard—and suggesting actions. Much work has already been done on explicitly encoding expert knowledge about domains into discourse system, so the issue is, what are the best machine learning methods for supporting discourse?

- **Natural learning.** This thesis has also emphasized that machine learning must be integrated seamlessly into the user interaction. Many learning systems today are just beginning to deal with how learning can take place naturally through the interface. Like any other system component, machine learning must be integrated so that the user does not feel that the whole purpose of the exercise is to train the machine but to get something done. Chatter is an attempt to integrate learning seamlessly into the user experience. What other kinds of learning methods can be fit into the interface in a natural way to the user?

- **Fast recognizer response times.** Chatter reinforces the need for large vocabulary recognizers to operate in real time so that user-computer interaction can be fluid and dynamic. Without good response times, the rhythm of natural conversation cannot be exploited. (This rule applies not only to the recognition subsystems but also to the rest of the feedback loop, including reasoning systems.) In this thesis, as much context as possible was used to pare down the active vocabulary so that recognition can be performed quickly and accurately.

9.3 Possible Future Directions

9.3.1 Other ways of learning

So far, Chatter's learning approach is best described as one by observation; it learns by watching what the user does and attempts to mimic what he may do in the future. While this technique appears seamless to the user, learning may progress more quickly if the agent actively seeks knowledge. As Maes and Kozierok [Mae93] have implemented, some more explicit interactive methods can be incorporated to make the agent learn more effectively:

- **Providing more explicit feedback.** If the interface informs the user of an interesting object, the user can vote on it positively or negatively, perhaps telling the agent what factors most influence the decision. The agent can learn about the importance of such factors more quickly rather than relying on more examples.

- **Being told.** The user can explicitly tell the agent to keep him informed of the state of affairs. For example, the user may have recently sent a message to someone from whom a reply is very important. The user can explicitly "program" through the interface by telling it to remind him of messages from that person. He might also want to tell the

interface when he is no longer so interested in messages from someone.

The challenge again is to integrate the mechanisms seamlessly into the interaction.

9.3.2 Integrating other applications

Chatter supports only access to information from, to and about people, yet supporting a work group involves many other types of data. An interesting application to cast into a conversational framework is the personal calendar. A conversational interface scales up well because the additional commands for the calendar and those pertaining to its integration with applications can be added transparently. The calendar has several major entities, including days, weeks and months, so the dialog becomes richer because the types of objects that can be discussed are greater.

From the discourse perspective, one of the most interesting challenges in this domain is to figure out how to manipulate and move objects occupying a space using speech. For example, imagine being able to say *move these appointments to that month, move it up to seven* or *move it up to the seventh*. It would be interesting to see how knowledge about the domain can be encoded and how anaphoric resolution can be implemented. From an agent perspective, calendars provide another abundant database of information. The agent can use the calendar as another basis for making suggestions. For instance, the agent may make inferences about the user's interests based on what he is doing. For example, if the calendar says that the user has an appointment with someone and there is a message from that person around the scheduled time, then the message may be more interesting than it would be otherwise. Alternatively, if the calendar records information about the user being away on a trip, then information about people at that locale or the locale itself (such as weather) may be considered more interesting than information from the user's home. Such decisions can be made on the basis of information in the calendar.

9.3.3 Representing negotiation

In Chatter, the specific tasks that users can ask their agent to perform are relatively simple and short-term—they are relatively simple to execute and they are begun and completed during the same session. Yet, what if the nature of our tasks require working with the computer to accomplish more complex tasks? For example, the user may negotiate with the agent on how best to deliver a message which was unsuccessfully delivered initially. In this case, the agent may have some ideas on how to accomplish the task but the user may have means more relevant to the current situation. The point is that the two may want to engage in a dialog to work out a plan which is acceptable to both. This type of discourse seems to

arise when the desired tasks occur over a longer time range and span multiple interaction sessions.

In [Sid92], Sidner proposes some ideas on how to represent collaborative conversations using a repertoire of proposals, acceptances, counter-proposals, rejections and so on. These actions have specific effects on the agent's representation of the user's beliefs and the state of the negotiation. They seem crucially useful when the domain of tasks become more complicated and the plans for execution are multi-faceted.

A. Grammar and Recognition Server

This appendix gives the grammar for the Chatter application as well as the API for the recognition server developed as part of this thesis.

A.1 Recognition Grammar

This section gives the grammar used in the Chatter application. The grammar is specified as a context-free grammar, and terminals are given in quotes. The grammar's format is given in a form readable for the recognizer supplied by TI. See [Wea92] for a more details about the format:

```
start(Default).
Default ---> CallPerson.
Default ---> ComposeMsg.
Default ---> ReadMsg.
Default ---> "good-bye".
Default ---> "pay" "attention".
Default ---> "stop" "listening".

CallPerson ---> CallCmd.
CallPerson ---> CallCmd Person.
CallCmd ---> "call".
CallCmd ---> "dial".

ComposeMsg ---> ComposeCmd.
ComposeMsg ---> ComposeCmd AMailObj.
ComposeMsg ---> ComposeCmd AMailObj Prep
Person.
ComposeMsg ---> ComposeCmd Person AMailObj.
ComposeCmd ---> "record".
ComposeCmd ---> "send".
ComposeCmd ---> "mail".
AMailObj ---> OptAMODet "email".
AMailObj ---> OptAMODet "email" "message".
AMailObj ---> "mail".
AMailObj ---> OptAMODet "message".
AMailObj ---> OptAMODet "voice mail".
OptAMODet ---> "an".
OptAMODet ---> "a".
OptAMODet ---> "".

ReadMsg ---> ReadCmd OptRMAdj "messages".
ReadMsg ---> OptRMQHeading OptRMMod "mes-
sages" OptRMSender.
ReadCmd ---> "read".
ReadCmd ---> "play".
OptRMAdj ---> "my".
OptRMAdj ---> "".
OptRMQHeading ---> "are" "there".
OptRMQHeading ---> "do" "I" "have".
OptRMMod ---> "any".
OptRMMod ---> "".
OptRMSender ---> "from" Person.
OptRMSender ---> "".

start(Read).
Read ---> Delete.
Read ---> Save.
Read ---> Reply.
Read ---> Forward.
Read ---> Scan.
Read ---> "what's" ReadInfo.
ReadInfo ---> "it" "about".
ReadInfo ---> "the" "subject".

Delete ---> "delete".
Delete ---> "delete" DMailObj.
Delete ---> "delete" PronRef.
DMailObj ---> OptDMODet "message".
DMailObj ---> OptDMODet "email" "message".
DMailObj ---> OptDMODet "voice" "mail".
PronRef ---> "it".
PronRef ---> "this".
PronRef ---> "that".
OptDMODet ---> "this".
OptDMODet ---> "that".
OptDMODet ---> "the".
OptDMODet ---> "".

Save ---> "save".
Save ---> "save" DMailObj.
Save ---> "save" PronRef.

Reply ---> "reply".
Reply ---> "reply" DMailObj.
Reply ---> "reply" "to" Person.
Reply ---> "reply" "to" PronRef.

Forward ---> "forward".
Forward ---> "forward" DMailObj "to" Per-
son.
Forward ---> "forward" "to" Person.

Scan ---> SLocation.
Scan ---> SLocation "message".
Scan ---> ScanCmd SMailObj.
SLocation ---> "next".
SLocation ---> "previous".
SLocation ---> "the" "last".
ScanCmd ---> "read".
ScanCmd ---> "what's".
SMailObj ---> "the" SLocation "message".

start(PersonInfo).
PersonInfo ---> PIQHeading OptPIMod Info.
```

```
PersonInfo ---> 'spell' OptPIMod Info.
PIQHeading ---> 'is' 'there'.
PIQHeading ---> 'what's'.
OptPIMod ---> 'a'.
OptPIMod ---> 'an'.
OptPIMod ---> 'the'.
OptPIMod ---> 'his'.
OptPIMod ---> 'her'.
OptPIMod ---> ''.
Info ---> 'address'.
Info ---> 'company'.
Info ---> 'email' 'address'.
Info ---> 'email'.
Info ---> 'fax' 'phone'.
Info ---> 'home' 'address'.
Info ---> 'home' 'phone'.
Info ---> 'name'.
Info ---> 'phone'.
Info ---> 'remarks'.
Info ---> 'work' 'address'.
Info ---> 'work' 'phone'.

start(ComposePerson).
ComposePerson ---> Person.
ComposePerson ---> Prep Person.

start(ComposeType).
ComposeType ---> 'email'.
ComposeType ---> 'voice' 'mail'.

start(ComposeFormat).
ComposeFormat ---> Format.
ComposeFormat ---> Format 'format'.

Format ---> 'mac'.
Format ---> 'macintosh'.
Format ---> 'mime'.
Format ---> 'next'.
Format ---> 'sun'.
Format ---> 'uuencode'.

start(Person).
Person ---> 'Atty'.
Person ---> 'Barry'.
Person ---> 'Chris'.
Person ---> 'Don'.
Person ---> 'Eric'.
Person ---> 'Gayle'.
Person ---> 'Jordan'.
Person ---> 'Lisa'.
Person ---> 'Lorin'.
Person ---> 'her'.
Person ---> 'him'.
Person ---> 'me'.

start(Prep).
Prep ---> 'for'.
Prep ---> 'from'.
Prep ---> 'of'.
Prep ---> 'to'.

start(Ack).
Ack ---> 'no'.
Ack ---> 'OK'.
Ack ---> 'yes'.
```

A.2 Recognition Server

This section outlines the current API of the recognition server, called *r_server*. Similar to the group's audio server, the recognition server cooperates with a client through a set of asynchronous function calls and callback routines. These functions are RPC stubs which communicate with the server. A client uses function calls to make requests to the server, such as starting recognition or training sessions. Acknowledgment of such requests are often returned as return values of the called functions. A client may also receive notification by the callback mechanism, which allows a client to be asynchronously notified of relevant events. The server protocol is implemented using the speech group's Socket Manager and Byte Stream Manager tools, which manage remote host connections and client-server RPC. See [Aro92] for more details on these utilities.

For connected-speech recognizers, no training needs to be done, so clients may simply begin recognition. However, most recognizers will want some kind of configuration file that gives the location of grammars, phonemes, and so on. For isolated-word recognizers, each connection to the server specifies a vocabulary of words to be recognized by the recognizer. Words are represented as nonnegative integers in the server, so the client is responsible for maintaining correspondences to the meaning of such indices. Vocabulary templates can be saved and loaded, and the server automatically provides this functionality to the client.

Calls to the recognition server are declared in <r_server.h>.

A.2.1 Initialization

A connection to the server is established or closed with these calls:

int r_open(const char *hostname, int recognizerType, const char *data)

Opens a connection to the recognition server and returns an associated file descriptor. *hostname* is the name of host on which to run the server; NULL may be specified for the current host. *recognizerType* is one of R_AGOG, R_IN3 or R_TISR. data is a recognizer-dependent argument. Pass in NULL by default. When using the TI recognizer, pass in the path to the configuration file.

int r_close(int fd)

Closes a connection with the server.

A.2.2 Requests

Once a connection has been opened, requests to the server can be made through the following set of calls:

void r_set_filename(int fd, char *vocabFile)

Sends the name of the vocabulary file (a file containing templates) or configuration file to the server.

void r_start_recognition(int fd)

Puts the recognizer in recognition mode. This function should be called just before a session of recognition.

void r_quit_recognition(int fd)

Takes the recognizer out of recognition mode.

void r_set_sex(int fd, int sex)

Sets the sex of the speaker. Specify either R_MALE or R_FEMALE. If this information can be provided by the client program, then recognition can be performed faster with connected-speech recognizers. This function only currently makes sense when using the TI

recognizer.

void r_subset(int fd, char *symbols)

Selectively enables the vocabulary for used in the recognizer. For isolated-word recognizers, *symbols* should be a colon-separated list of index numbers that are to be active. For connected-speech recognizers, *symbols* should be a colon-separated list of the start hypotheses.

The following functions applies to speaker-dependent recognizers only.

void r_disk_to_recognizer(int fd)

Loads the voice template file of the given user and vocabulary name from server disk to the recognizer's active memory. If the named vocabulary does not exist, –1 is returned.

void r_recognizer_to_disk(int fd)

Saves the current template file to server disk under the current name specified by r_set_filename. This command must be called to save any exchanges or updates that have been made.

void r_start_training(int fd)

Call this just before calling r_train to initiate a training session.

void r_train(int fd, int wordNum)

Begin a training session on word *wordNum*. If a template at *wordNum* does not already exist, then one will be created. Otherwise, the existing template is refined. This call will generate R_BEGIN_TRAIN_EV and R_END_TRAIN_EV events. One or more R_BEGIN_TRAIN_EV events will be generated to begin training and refine the template, and R_END_TRAIN_EV will be sent upon completion of training the word. Before calling r_train, you must call r_start_training first.

void r_quit_training(int fd)

Called to finish a training session. A call to r_start_training must eventually end with r_quit_training.

int r_main_loop()

Invokes socket manager's SmMainLoop to run asynchronously and poll for events.

A.2.3 Notifications

To receive events generated by the server, a client must implement a set of callback functions and register them with the server. Recognition results may be a string indicating the utterance most likely spoken or an index into a vocabulary list, or it may even be a set of these ranked in order from most confident prediction to least. Associated with these results may be some confidence measures, so the results to be returned may be quite structured.

*void r_recognize_ev(int fd, void *clientdata, char *string)*

*void r_recognize_ev(int fd, void *clientdata, int firstWord)*

When a sentence or word is recognized in recognition mode, one of these callbacks will be invoked with the recognition result. The first version is used to return results from connected-speech recognizers, while the second is used for isolated-word recognizers.

*void r_recognize_plus_ev(int fd, void *clientdata, char *firstString, int firstScore, char *secondString, int secondScore)*

*void r_recognize_plus_ev(int fd, void *clientdata, int firstWord, int firstScore, int secondWord, int secondScore)*

This callback differs only slightly from r_recognize_ev in that it provides more statistics about the word recognized. Whenever r_recognize_ev is called, so is r_recognize_plus_ev, and vice versa, if so registered. Again, there are two versions, similar to the r_recognize_ev callback.

void r_begin_train_ev(int fd, int data, int wordNum)

When training is initiated, a training "begin" event will be generated each time a training template is expected from the user. This callback allows the client to perform appropriate output to prompt the user to speak. Note that this event may be generated several times for one word, since many recognizer require multiple training sessions to generate a reliable template. Only recognizers that require training will generate these events.

void r_end_train_ev(int fd, int data, int wordNum)

When training for a word has completed, this callback will be called. Unlike begin_train_ev, this callback will only be called once for each word. If a sequence of words is to be trained during the training session, this callback can begin another training session using r_train.

Callbacks can be registered and unregistered by the calls:

int r_register_callback(char *event, (void *cb)(), void *clientData)

event is one of: `R_BEGIN_TRAIN_EV`, `R_END_TRAIN_EV`, `R_RECOGNIZE_EV`, or `R_RECOGNIZE_PLUS_EV`.

int r_unregister_callback(char *event, (void *cb)())

Give the associated callback function for which you gave when you registered the callback.

B. Event Server

Appendix B describes the API for the *Zeus* event server. See Chapter 4 on page 37 for a more general description of *Zeus*. This server serves as a repository for information about people. It is intended to be quite simple in design; the "smarts" for interpreting and using the event information will reside in client peripheral processes. We make the distinction between *client* and *daemon* processes. Roughly, *clients* are processes which use the information in the server, while *daemons* are processes which help maintain information in the server. The distinction is not actually critical because both access the server in the same way.

The structure of the information in *Zeus* is hierarchical, rather like the UNIX file system. There are two types of nodes in the hierarchy—folders and records. Folders have as their contents sets of other folders and records. Records have as their contents some user-defined content, which will initially be plain text. Places in the hierarchy will be accessed using the same convention for UNIX files, for example `/bar/foo` to get to `foo` in folder `bar`.

Notification is implemented as follows. Processes can register a daemon for a given folder, and whenever information in the folder changes, they informed of the change. Note that such notified daemons might cause further changes in the event server, so there may be several iterations of notification.

Although *Zeus* implements the client-server connection using Socket Manager and Byte Stream Manager libraries, it is more convenient to representation the connection on both the client and server sides as C++ objects. These classes also provide abstractions so that dtype objects can be easily shipped between the server and client through lower-level transport mechanism; they automatically convert dtypes to and from binary representations.

In the next section, we describe the API for communicating with the event server and for writing clients.

B.1 Event Server Classes

The API for communicating with *Zeus* is organized around a C++ class called ZConnection. To establish a connection to the server, one creates a new instance of ZConnection. Messages to a ZConnection object then change the state of the event server.

B.1.1 SConnection

ZConnection is a subclass of SConnection, an abstract superclass for implementing client connections to servers. An SConnection maintains the connection as implemented by the Socket Manager and Byte Stream Manager, including built-in facilities for handling errors or failures. An SConnection object is persistent in that if the server goes down, the object will attempt to re-establish connection to the server periodically. This period degrades exponentially. Here is the class description:

```
class SConnection
{
  public:
    int connfd;
    SConnection(const char *server, const char *hostname);
    virtual ~SConnection();
    virtual int mainLoop();
    virtual int close();

    virtual BOOL isUp();
    virtual void connectionDown();
    virtual void connectionUp();
    virtual void registerCB(int type, SConnectionCB cb, void *data);
    virtual void unregisterCB(int type, SConnectionCB cb);

  protected:
    SmTimer *timer;
    int timeout_ms;
    llist *downCBs;
    llist *upCBs;
    char *serviceName;
    char *host;
};
```

int connfd

Gives the file descriptor of the current connection to the event server. If this variable is less than 0, then it indicates that the event server is down.

SConnection(const char *server, const char *hostname)

Constructor for the class. *server* specifies the service name and *hostname* gives the host of the service.

virtual ~SConnection()

Closes the connection to the server and destroys memory for the object.

virtual int mainLoop()

Executes the Socket Manager's SmMainLoop function. This method does not return until a close has been executed.

virtual int close()

Closes down the connection to the server.

virtual BOOL isUp()

Returns YES if the server is up and there is a connection to the event server.

virtual void connectionDown()

This method is invoked in the case that the event server goes down. By default, this method does nothing, but it can be overridden by a subclass to do whatever operations are appropriate to keep the state of the client from going awry when the server goes down.

virtual void connectionUp()

This method is invoked as soon as the server comes up after it has been down. By default, this method also does nothing.

virtual void registerCB(int type, SConnectionCB cb, void *data)

Registers a callback with the object to be notified when the server connection goes down and/or up or some error condition with the connection. Currently, *type* can be either SC_DOWN or SC_UP, and *cb* is the callback that will be invoked when the specified event occurs. *data* is a pointer to client data that will be passed in as an argument of the callback. An arbitrary number of callbacks can be given for these two event types.

The callback function should be in the form specified by the SConnectionCB type:

```
typedef void (*SConnectionCB)(SConnection *, void *)
```

When invoked, the function will receive the associated SConnection object where the event occurred and client data given using this function.

virtual void unregisterCB(int type, SConnectionCB cb)

Unregisters a callback previously registered with registerCB.

B.1.2 ZConnection

The ZConnection class is a client's representation of a connection to the event server. Creating an instance of ZConnection forms a new connection to the event server, and messages sent to the object result in changes in the contents of the server. This class is declared in <ZConnection.hh>.

```
class ZConnection : public SConnection
{
  public:
    ZConnection(const char *hostname = "zeus");
    virtual ~ZConnection();

    virtual int close();
    virtual void connectionUp();

    virtual BOOL exists(const char *path);
    virtual dtype *getFolder(const char *path);
    virtual dtype *getFolderContents(const char *path);
    virtual void newFolder(const char *newPath);
    virtual void deleteFolder(const char *path);
    virtual dtype *getRecord(const char *path);
    virtual void newRecord(const char *newPath, dtype *newContents);
    virtual void changeRecord(const char *path, dtype *newContents);
    virtual void deleteRecord(const char *path);

    virtual void beginUpdate(const char *path);
    virtual void endUpdate(const char *path);

    virtual void registerDaemon(const char *path, void (*cb)(),
                                void *data);
    virtual void unregisterDaemon(const char *path);

    virtual dtype *callbacks();

  protected:
    dtype *cbs;
};
```

ZConnection(const char *hostname = "zeus")

Constructor for ZConnection class. Note that the **hostname** argument need not be specified and defaults to the host zeus.

virtual ~ZConnection()

Closes the connection to the event server and deallocates the memory associated with the connection.

virtual int close()

Overrides the superclass's implementation of close to unregister any daemons that were not explicitly unregistered before this method is invoked.

virtual void connectionUp()

Overrides the superclass's implementation to re-register the client's previously registered daemons with the server after the server comes back online.

virtual BOOL exists(const char *path)

Returns YES if the named path exists.

virtual dtype *getFolder(const char *path)

Returns a dt_list of the sub-folder and record names of the named folder. The list consists of dt_strings giving the complete paths of the information in the folder. For instance, invoking the method with `/user/mullins`, will return `("/user/mullins/e-mail" "/user/mullins/vmail" "/user/mullins/activity" )`. The result should be deleted after use by the caller. If the folder does not exist or the server connection is down, NULL is returned.

virtual dtype *getFolderContents(const char *path)

Returns a dt_list containing the complete contents of all records and folders in the named folder. If there are folders embedded in the given path name, then their complete contents are returned as well. Note that the names of the records and folders in the folder are not returned, so this method is only of limited use. This method is useful when the structure of a folder's contents is simple and the contents need to be retrieved for processing. One way to get good use out of this method is to represent records in such a way that the names of the records can be easily derived from their contents, perhaps as some field in the record.

The result should be freed after use. If the folder does not exist, NULL is returned.

virtual void newFolder(const char *newPath)

Creates a new folder at the named path. The path must be given absolutely, and all folder names leading to the last one in the path must already exist. If the path already exists in the server, no changes are made.

virtual void deleteFolder(const char *path)

Removes the folder at the named path including any contents in the folder. No changes are made if the path does not exist.

virtual dtype *getRecord(const char *path)

Returns the contents of the record at the named path. The contents were placed as a result of **newRecord** or **changeRecord** methods. The path should refer to a record. The result should be deleted after use. NULL is returned if the record does not exist.

virtual void newRecord(const char *newPath, dtype *newContents)

Adds a new record with the given named path and contents to the server. **newContents** may be any valid **dtype**. If any daemons are registered with the immediately enclosing folder, a Z_NEW event is generated for all such daemons for the enclosing folder. All folder names leading up to the last name must already exist. The caller is responsible for deleting **newContents** afterwards.

virtual void changeRecord(const char *path, dtype *newContents)

Changes the contents of the record at the named path with the newly-given contents. If any daemons are registered with the immediately enclosing folder, a Z_CHANGED event is generated for such daemons for the enclosing folder. If the named record does not exist, then no changes are made. The caller is responsible for deleting **newContents** afterwards.

virtual void deleteRecord(const char *path)

Deletes the record at the named path. A Z_DELETED event is generated if any daemons are registered for the enclosing folder.

virtual void beginUpdate(const char *path)

Informs the server that the caller will be making several changes to the named folder given by the path. The server will buffer all changes and suspend notification to daemons registered for this folder until a matching **endUpdate** is encountered. Since multiple clients may be making changes to the server, **beginUpdates** may be nested and daemon notification won't occur until the last nested **endUpdate** is invoked.

Normally, any change to the contents of a folder will result in notification events, and multiple changes resulting in multiple notifications may be inefficient, so clients expect-

ing to make several changes to a folder should invoke this method before performing updates.

virtual void endUpdate(const char *path)

Tells the server that the client has finished updating the named folder. The server will notify any daemons of all the changes since the first beginUpdate.

virtual void registerDaemon(const char *path, void (*cb)(), void *data)

Registers a callback for the named folder so that when any changes are made to the records of the folder, the callback is invoked. The caller may supply client data that will be passed as an argument to the callback. See above for details on the format of the callback. Note that there is a restriction of one callback per folder per client.

void report(ZConnection *server, void *clientdata, dtype *howChanged, dtype *contents)

virtual void unregisterDaemon(const char *path)

Unregisters a previously registered callback for the named folder.

B.1.3 A simple example

This section gives a simple example of how these classes can be used to communicate with the event server. This program gets records of all new messages for the user `lisa`.

```
#include <ZConnection.hh>
#include <sys/param.h>

int main()
{
    ZConnection *server;
    dtype *msgs;
    char path[MAXPATHLEN];

    server = new ZConnection;
    Z_MAKE_USER_PATH(path, "lisa", Z_EMAIL);
    msgs = server->getFolderContents(path);
    cout << "Records of Lisa's new messages are:\n\n" << msgs;
    delete msgs;
    server->close();
    delete server;

    return 0;
}
```

A new ZConnection is created, and a path to the folder `/user/lisa/email` is created. Its

contents are retrieved and printed out to the terminal. Finally, the server connection is closed and the server object is freed.

B.2 Information Contents

The specification for the server makes no mention of what are stored as events. This is intended so that the information in the server can augmented or changed in the future with little trouble. Currently, this information is being kept in the server: users' activity and location, new e-mail and new voice mail. The following constants are declared in `<zinfo.hh>`.

B.2.1 Activity Information

A call to `Z_MAKE_USER_PATH(path, user, Z_ACTIVITY)` calculates the path to *user*'s activity record. Similarly, a call to `Z_MAKE_USER_PATH(path, user, Z_ALERT)` obtains a folder of names who have placed alerts on the given user. An activity record has the following field. See the activity server header files for more information about the meaning of these fields.

`Z_AS_PERSON`	Login name of the person
`Z_AS_PLACE`	Person's location
`Z_AS_TIME`	String version of time last updated
`Z_AS_STATE`	User's state
`Z_AS_PHONE`	Number of the user's nearest phone
`Z_AS_HOST`	Workstation where the user is logged on
`Z_AS_NAME`	User's real name
`Z_AS_HOST_STATE`	State of the person's workstation
`Z_AS_TIMESTAMP`	Last time the information was updated

B.2.2 E-mail Information

A call to `Z_MAKE_USER_PATH(path, user, Z_EMAIL)` obtains the folder of records representing new mail messages. The following fields in those records are always present:

`Z_EMAIL_DATE`	Time the message was received
`Z_EMAIL_FROM`	E-mail address of the sender
`Z_EMAIL_TO`	Who the mail is addressed to
`Z_EMAIL_SUBJECT`	Subject of the message
`Z_EMAIL_HEADER_OFFSET`	Offset to the message's header in the spool file
`Z_EMAIL_BODY_OFFSET`	Offset to the message's body in the spool file

Z_EMAIL_BODY_LENGTH	The number of characters in the message body

These fields are optional in a record:

Z_EMAIL_CC	Contents of the Cc: header if one was available
Z_EMAIL_NAME	Real name of the message's sender
Z_EMAIL_FORMAT	Multimedia mail format if the message was such
Z_EMAIL_PRONOUNCE	procmail filter category

B.2.3 Voice mail information

Similarly, a call to Z_MAKE_USER_PATH(*path*, *user*, Z_VMAIL) obtains the folder of records representing new voice mail messages. A voice mail record always has these fields:

Z_VMAIL_DATE	Time the message was received
Z_VMAIL_FROM	String giving sender of the message
Z_VMAIL_FILE	Path to the message's audio file
Z_VMAIL_LENGTH	Length in seconds of the audio file

These are optional fields of a voice mail record:

Z_VMAIL_PHONE	Phone number of caller if available

B.2.4 Other constants and macros

Two other constants are defined:

Z_USER_PATH	Gives the path to the folder holding all of the users' information. For example, the message getFolder(Z_USER_PATH) gives the names of all users registered with the server.
Z_USER	Gives the path to the folder holding all of a given user's information. For instance, a Z_MAKE_USER_PATH(*path*, *user*, Z_USER) followed by a getFolder(*path*) will return a listing of the information stored for the user.

In addition, these macros can be used to create strings of the appropriate paths leading to information about a user in the server.

void Z_MAKE_USER_PATH(char *path, const char *user, const char *pTemplate)

Used to construct a path leading to some given user's information. *pTemplate* is one of the templates given above. The result in placed in *path*.

void Z_MAKE_USER_RECORD(char *path, const char *user, const char *pTemplate, const char *recordName)

Used to construct a path leading to some given user's record when the information is stored as a folder of records. Here, *recordName* specifies the actual name of the record to be access. For example, `Z_MAKE_USER_RECORD(path, "caesar", Z_ALERT, "lisa")` obtains a path for the record `/user/caesar/alerts/lisa`.

void Z_GET_USER_FROM_PATH(char *username, const char *path)

Given any valid path, this macro returns the user associated with the path in *username*.

B.3 A Class for Implementing *Zeus* Clients

If clients are implemented as classes, then it is often desirable to have several clients operating in the same process sharing the same connection to the event server. A ZeusClient is designed to be subclassed for the implementation of such clients. A ZeusClient allows several clients operating in the same process to share a common connection to the event server.

```
class ZeusClient
{
  public:
    ZeusClient(const char *user = NULL, ZConnection *connection = NULL);
    virtual ~ZeusClient();

    virtual ZConnection *connection();
    virtual void connectionDown();
    virtual void connectionUp();

  protected:
    BOOL freeZConnection;
    char *login;
    ZConnection *zserver;
};
```

ZeusClient(const char *user = NULL, ZConnection *connection = NULL)

Constructor for the class. Both arguments to this method are optional. *user* gives the login name of the current process. If it is not supplied or NULL, the owner of the current process is used. If *connection* is empty, then a new ZConnection is automatically created. If one is passed in to the constructor, then it is assumed that the connection object is shared, and it will not be freed when ZeusClient is freed.

virtual ~ZeusClient()

Destructor for the class. Frees the object's ZConnection object if it was created by the

object's constructor.

virtual ZConnection *connection()

Returns the ZConnection being used by this ZeusClient.

virtual void connectionDown()

This method is invoked when the object's connection to the server goes down. By default, this method does nothing but is meant to be subclassed so that necessary actions can be taken when the connection goes down.

virtual void connectionUp()

Similar to connectionDown, this method is invoked when the server comes back up. By default, it does nothing.

In Chatter, all clients that communicate with *Zeus* are implemented as subclasses of ZeusClient.

C. Class Hierarchies

Appendix C outlines the organization and implementation of Chatter and its supporting environment. Shown are the C++ class inheritance hierarchies that are implemented. The first hierarchy shows internal recognition server architecture detailed in Chapter 8 on page 90.

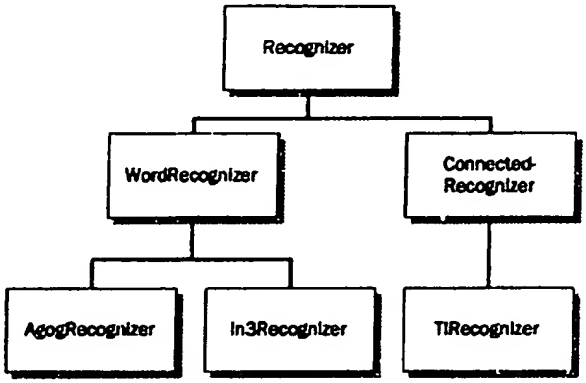

Figure C.1: Recognizer class hierarchy.

The tree below represents the discourse system architecture. See "Choosing appropriate segments" on page 59 for details on the functionality of the segments.

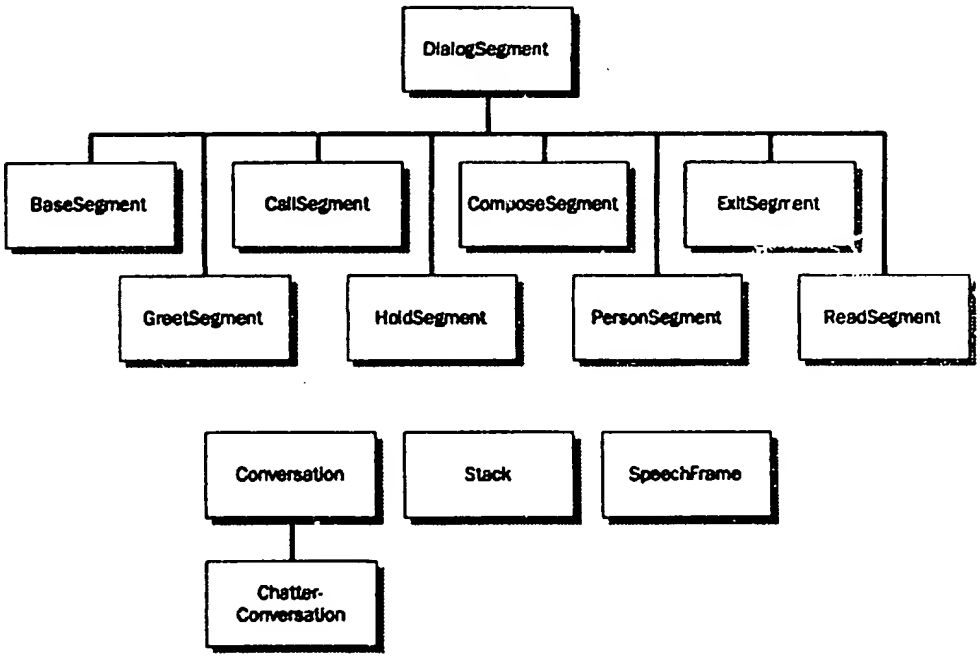

Figure C.2: Chatter discourse management class hierarchy.

These hierarchies show the organization of the *Zeus* event server and its clients, which is described in Chapter 4 on page 41.

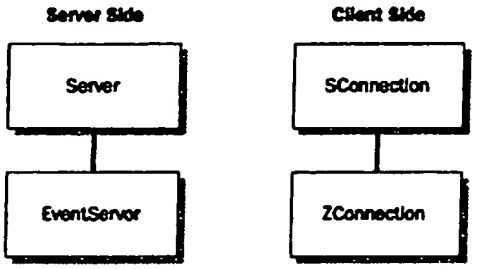

Figure C.3: *Zeus* server and client hierarchies.

Event server clients for Chatter are shown below. See Chapters 4 (on page 41) and 7 (on page 73) for a description of these clients.

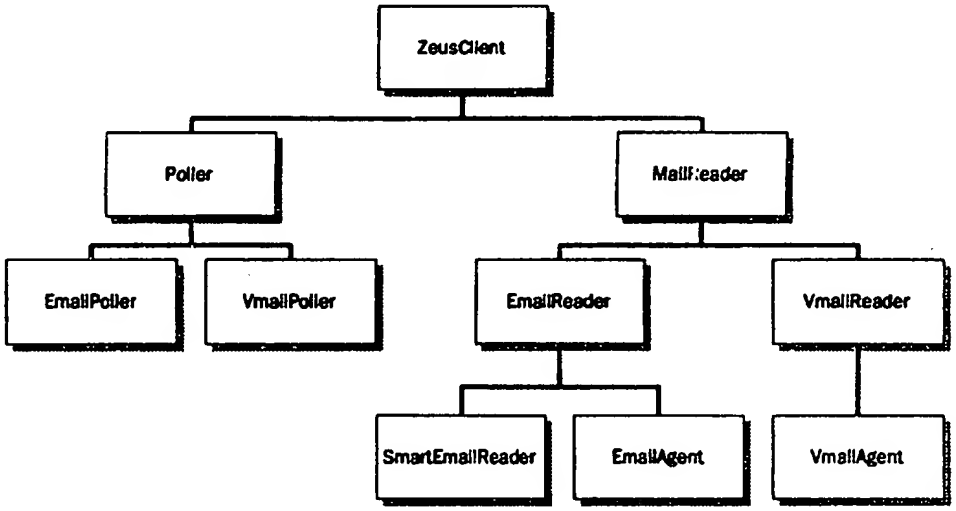

Figure C.4: *Zeus* clients hierarchy.

Finally, the MBRStore class is shown. Its description begins on page 74.

Figure C.5: MBRStore class.

References

[Abr92] N. Abramson. dtype++—Uniification of Commonly Used Data Types. Electronic Publishing Group, MIT Media Laboratory, 1992.

[Aho88] A. Aho, R. Sethi and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1988.

[Aro91] B. Arons. Hyperspeech: Navigating in Speech-Only Hypermedia. *In proceedings of* Hypertext '91, pp. 133–146. ACM, 1991.

[Aro92] B. Arons. Tools for Building Asynchronous Servers to Support Speech and Audio Applications. *Proceedings of the* ACM Symposium on User Interface Software and Technology, 1992.

[Bra84] M. Brady and R. Berwick. *Computational Models of Discourse.* MIT Press, 1984.

[Cah92] J. E. Cahn. A Computational Architecture for Mutual Understanding in Dialog. MIT Media Laboratory Technical Report 92-4, 1992.

[Cah92a] J. E. Cahn, An Investigation into the Correlation of Cue Phrases, Unfilled Pauses and the Structuring of Spoken Discourse. *In Proceedings of* the IRCS Workshop on Prosody in Natural Speech. University of Pennsylvania, 1992.

[Cha91] B. Chalfonte, R. Fish and R. Kraut. Expressive Richness: A Comparison of Speech and Text as Media for Revision. *In proceedings of* Conference on Computer Human Interaction, pp. 21–26, 1991.

[Chi92] D. Chin. Intelligent Interfaces as Agents. *Intelligent User Interfaces*, J. Sullivan and S. Tyler (eds), pp. 177–206, ACM Press, 1992.

[Cla87] H. Clark and E. Schaefer. Collaborating on Contributions to Conversations. *Language and Cognitive Processes*, 2:1–23, 1987.

[Cla89] H. Clark and E. Schaefer. Contributing to Discourse. In *Cognitive Science*, 13:259–294.

[DeJ86] G. DeJong and R. Mooney. Explanation-Based Learning: An Alternative View. *Machine Learning 1*, 2, pp. 145–176, April 1986.

[Haa92] K. Haase. FRAMER, A Database System for Knowledge Representation and Media Annotation. MIT Media Laboratory Technical Note 92-5, 1992.

[Her87] G. Herman, M. Ordun, C. Riley and L. Woodbury. The Modular Integrated Communications Environment (MICE): A System for Prototyping and Evaluating Communication Services. *In proceedings of* 1987 International Switching

Symposium, pp. 442–447, 1987.

[Hir73] L. Hirschman. Female-Male Differences in Conversational Interaction. Presented at the annual LSA meeting, San Diego, 1973.

[Hir91] L. Hirschman, S. Seneff, D. Goodine and M. Phillips. Integrating Syntax and Semantics into Spoken Language Understanding. *In proceedings of* Fourth DARPA Speech and Natural Language Workshop, 1991.

[Jel85] f. Jelinek. The Development of an Experimental Discrete Dictation Recognizer. *In* Proceedings of the IEEE, vol. 73, pp. 1616–1624, 1985.

[Goo92] D. Goodine, L. Hirschman, J. Polifroni, S. Seneff and V. Zue. Evaluating Interactive Spoken Language Systems. *In proceedings of* Second International Conference on Spoken Language Processing, 1992.

[Gro86] B. Grosz and C. Sidner. Attention, Intentions, and the Structure of Discourse. *Computational Linguistics*, 12(3):175–203, 1986.

[Gro89] B. Grosz, M. Pollack and C. Sidner. Discourse. *Foundations of Cognitive Science*. M. Posner (ed), pp. 437–468, MIT Press, 1989.

[Gro90] B. Grosz and C. Sidner. Plans for Discourse. *Intentions and Communication*. P. Cohen, Morgan and M. Pollack (eds), pp. 105–133, MIT Press, 1990.

[Kam90] R. Kamel, K. Emami, R. Eckert. PX: Supporting Voice in Workstations. *IEEE Computer*, 23(8):73–80, 1990.

[Kau90] H. Kautz. A Circumscriptive Theory of Plan Recognition. *Intentions and Communication*. P. Cohen, Morgan and M. Pollack (eds), pp. 105–133, MIT Press, 1990.

[Kap73] R. Kaplan. A General Syntactic Processor. *Natural Language Processing*. Randall Rustin (ed), pp. 193–241, Algorithmics Press, 1973.

[Kay67] M. Kay. Experiments with a Powerful Parser. *In proceedings of* 2nd International COLING Conference #10, 1967.

[Kay90] A. Kay. User Interface: A Personal View. *The Art of Human-Computer Interface Design*, B. Laurel (ed), Addison-Wesley, 1990.

[Koz93] R. Kozierok and P. Maes. A Learning Interface Agent for Scheduling Meetings. *In proceedings of* the ACM-SIGCHI International Workshop on Intelligent User Interfaces, 1993.

[Koz93a] R. Kozierok. A Learning Approach to Knowledge Acquisition for Intelligent Interface Agents. MIT Masters Thesis, Department of Electrical Engineering and Computer Science, 1993.

[Lau90] B. Laurel. Interface Agents: Metaphors with Character. *The Art of Human-Computer Interface Design*, B. Laurel (ed), Addison-Wesley, 1990.

[Loc90] K. Lochbaum, B. Grosz and C. Sidner. Models of Plans to Support Communication: An Initial Report. *In proceedings of* AAAI-90, 1990.

[Lövstrand. Being Selectively Aware with the Khronika System. *In proceedings of* ECSCW-91, 1991.

[Mae93] P. Maes and R. Kozierok. Learning Interface Agents. *Submitted to* AAAI-93, National Conference on Artificial Intelligence, 1993.

[Mal87] T. Malone, R. Grant, K-Y. Lai, R. Rao and D. Rosenblitt. The Information Lens: An Intelligent System for Information Sharing and Coordination. *Technological Support for Work Group Collaboration*, M. Olson (ed), 1989.

[Mas90] T. Mason and D. Brown. *lex & yacc.* D. Dougherty (ed), O'Reilly and Associates, 1990.

[Mic83] R. Michalski, J. Carbonell and T. Mitchell. *Machine Learning.* Tioga Press, 1983.

[Mul90] M. Muller and J. Daniel. Toward a Definition of Voice Documents. *In proceedings of* COIS '90, pp. 174–182. ACM, 190.

[Mul92] A. Mullins. The Design and Implementation of the HyperNews System. Term paper, MIT Media Laboratory, (mullins@media-lab.mit.edu).

[Mye91] B. Myers. Demonstrational Interfaces: Coming Soon? *Panel at* ACM CHI-91, 1991.

[Neg90] N. Negroponte. Hospital Corners. *The Art of Human-Computer Interface Design*, B. Laurel (ed), Addison-Wesley, 1990.

[Orw93] J. Orwant. Doppelgänger Goes To School: Machine Learning for User Modeling. MIT Masters Thesis, Media Arts and Sciences Program, 1993.

[Ovi92] S. Oviatt and P. Cohen. The Contributing Influence of Speech and Interaction on Human Discourse Patterns. *Intelligent User Interfaces*, J. Sullican and S. Tyler (eds), pp. 69–84, ACM Press, 1992.

[Pis85] D. B. Pisoni, H. C. Nusbaum and B. G. Greene. Constraints on the Perception of Synthetic Speech Generated by Rule. Behavior Research Methods, Instruments & Computers, 17(2):235–242, 1985.

[Rud91] A. Rudnicky and A. Hauptmann. Models for Evaluating Interaction Protocols in Speech Recognition. *In proceedings of* CHI '91, pp.285–291. ACM, 1991.

[Rut87] D. Rutter. *Communicating by Telephone.* Pergamon Press, 1987.

[Sch82] R. Schank. *Dynamic Memory: A Theory of Reminding and Learning in Computers and People.* Cambridge University Press, 1982.

[Sch86] C. Schmandt and B. Arons. A Robust Parser and Dialog Generator for a Con-

versational Office System. *In proceedings of* American Voice Input/Output Society, 1986.

[Sch84] C. Schmandt and B. Arons. A Conversation Telephone Messaging System. IEEE Transactions on Consumer Electronics, CE-30(3):xxi-xxiv, 1984.

[Sch85] C. Schmandt and B. Arons. Phone Slave: A Graphic Telecommunications Interface. *Proceedings of the Society for Information Display*, 26(1):79–82, 1985.

[Sch89] C. Schmandt and S. Casner. Phonetool: Integrating Telephones and Workstations. *In proceedings of* GLOBECOM '89. IEEE Communications Society, 1989.

[Sch90] C. Schmandt. Caltalk: A Multimedia Calendar. *In proceedings of* American Voice Input/Output Society, 1990.

[Sch93] C. Schmandt. Phoneshell: The Telephone as Computer Terminal. *To appear in* Proceedings of ACM Multimedia '93 Conference, 1993.

[Sej86] T. J. Sejnowski and C. R. Rosenberg. NETtalk: A Parallel Network that Learns to Read Aloud.Johns Hopkins University Technical Report, JHU/EECS-86/01, 1986.

[Sen91] S. Seneff, L. Hirschman and V. Zue. Interactive Problem Solving and Dialogue in the ATIS Domain. *In proceedings of* Fourth DARPA Speech and Natural Language Workshop, 1991.

[Sid92] C. Sidner. Using Discourse to Negotiate in Collaborative Activity: An Artificial Language. *In proceedings of workshop on* Cooperation Among Heterogeneous Agents, NCAI-92.

[Sta86] C. Stanfill and D. Waltz. Toward Memory-Based Reasoning. *Communications of the ACM*, 29(12): 1213–1228, 1986.

[Sti91] L. Stifelman. Not Just Another Voice Mail System. *In proceedings of* American Voice Input/Output Society, pp. 21 26, 1991.

[Sti92] L. Stifelman. VoiceNotes: An Application for a Voice-Controlled Hand-Held Computer. MIT Masters Thesis, Media Arts and Sciences Program, 1992.

[Sti93] L. Stifelman, B. Arons, C. Schmandt and E. Hulteen. VoiceNotes: A Speech Interface for a Hand-Held Voice Notetaker. *In proceedings of* INTERCHI '93, 1993.

[Van93] S.R. van den Berg. Procmail program. *Available from* ftp.informatik.rwth-aachen.de (137.226.112.172), 1993.

[Wah92] W. Wahlster. User and Discourse Models for Multimodal Communication. *Intelligent User Interfaces*, J. Sullivan and S. Tyler (eds), pp. 45–68, ACM Press, 1992.

[Wan92] R. Want and A. Hopper. Active Badges and Personal Interactive Computing Objects. *IEEE Transactions on Consumer Electronics*, 38(1):10–20, 1992.

[Wat84] J. Waterworth. Interaction with Machines by Voice: A Telecommunications Persepctive. *Behaviour and Information Technology*, 3(2):163–177, 1984.

[Whe92] B. Wheatley, J. Tadlock and C. Hemphill. Automatic Efficiency Improvements for Telecommunications Application Grammars. Texas Instruments technical report.

[Wil77] C. Wilson and E. Williams. Watergate Words: A Naturalistic Study of Media and Communications. *Communications Research*, 4(2):169–178, 1977.

[Won91] C. C. Wong. Personal Communications. MIT Masters Thesis, Media Arts and Sciences Program, 1991.

[Zel88] P. Zellweger, D. Terry and D. Swinehart. An Overview of the Etherphone System and its Applications. *In proceedings of* 2nd IEEE Conference on Computer Workstations, pp. 160–168, 1988.

[Zue89] V. Zue, J. Glass, D. Goodine, H. Leung, M. Phillips, J. Polifroni and S. Seneff. The Voyager Speech Understanding System: A Progress Report. *In proceedings of* Second DARPA Speech and Natural Language Workshop, 1989.

[Zue90] V. Zue, J. Glass, D. Goddeau, D. Goodine, H. Leung, M. McCandless, M. Phillips, J. Polifroni, S. Seneff and D. Whitney. Recent Progress on the MIT Voyager Spoken Language System. *In proceedings of* International conference on Spoken Language Processing, 1990.